

Programmer's Guide

M2Sprint 
The integrated Modula-2 environment for Commodore-Amiga™ computers

*Developed by M2S Inc.
Dallas, TX*

Printed in the USA

All products mentioned in this manual are trademarks of their respective owners.

Copyright 1989 M2S Inc. All rights reserved.

No part of this publication may be copied or distributed, transmitted, transcribed or translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, manual, or otherwise, or disclosed to third parties without the express written permission of M2S Inc.

M2S Inc.
P.O. Box 550279
Dallas, TX 75355

Preface

This manual was created entirely on the Amiga using the following products:

- *Professional Page* v1.1 from Gold Disk Inc.
- *Grabbit!* from Discovery Software
- *M2E* from M2S
- *Deluxe Paint II* from Electronic Arts

Laser printers used include General Computing Corp.'s *Business Laserprinter* and Apple's *LaserWriter*.

Credits

- Documentation and layout by Brian Dueck
- Compiler and linker by Martin Fisher
- Editor, support programs and utilities, library modules and additional documentation by Martin Taillefer
- M2Sprint project manager, Phil Camp

Many thanks to the group of M2Sprint beta-testers who's vigilant bug-bashing and intuitive suggestions helped to polish this product.

The Compiler, Linker, Editor, support programs and utilities were created entirely with M2Sprint.

Dedication

This manual and initial version of M2Sprint is dedicated to the memory of Arnaud Rubin, the 28 year old M2Sprint beta-tester who lost his life in the tragic Pan-Am flight 103 air disaster over Lockerbee, Scotland on December 21st 1988.

Arnaud's day job focused on developing artificial intelligence software, and his enthusiasm for computers lead him to be a dedicated supporter of the Amiga during his off hours.

Living in Belgium, Arnaud spent much of his free time learning about the Amiga by participating in networks like BIX, CompuServe, and PeopleLink. He was always happy to pass his knowledge on to others, and as such played an active role in local Amiga user's groups. The contents of this M2Sprint product were certainly influenced by the insightful proposals and suggestions that Arnaud, as a beta-tester, contributed.

He will be sorely missed by friends, family, and the community of Amiga users who knew him.

Table of Contents

I. Welcome!

1. M2Sprint and the Amiga
2. Reading the Programmer's Guide
 - Making the most of your time 2-2
 - Manual conventions 2-4
3. Contacting M2S

II. Getting Started

4. System requirements
5. Before you begin
 - What's in the box? 5-2
 - Making backups 5-3
 - Warranty registration 5-7
6. Installing M2Sprint
 - Using M2Install 6-2
7. Customizing the environment
 - Logical assignments 7-2
 - Customization suggestions 7-5
 - Resident mode commands 7-6

III. Understanding M2Sprint

8. Using the environment
 - Text editing with M2E 8-2

Compiling with M2C	8-5
Finding bugs with M2Errors and M2E	8-7
Linking with M2L	8-9
Running "HelloWorld"	8-11

9. Library overview

Amiga system interface	9-2
Standard Modula-2 function	9-3
Easy Amiga interface	9-4
'C' language support	9-7
IFF interface	9-8
ARP interface	9-9
ARexx macro language interface	9-10
Amiga utility modules	9-11

10. Using the libraries

General	10-2
Modula-2 and C	10-3
Exec	10-4
Libraries	10-8
Devices	10-10

IV. M2Sprint Components

11. M2E - The Editor

Launching M2E	11-2
The M2E display window	11-4
File Management	11-6
Cursor positioning and scrolling	11-10
Editing	11-12
Configuration	11-17
Advanced features	11-19
Command summary	11-23
The file requester	11-27
Menu summary	11-29

12. M2C - The Compiler

Compilation errors	12-3
Version control	12-4
Compiler options	12-5

Compiler switches	12-6
The SYSTEM module	12-9
Data types	12-16
Extensions and restrictions	12-19
13. M2L - The Linker	
Using M2L	13-2
Locating IMPORTed modules	13-4
Run time support	13-5
14. Support programs and utilities	
M2FastLoad	14-2
M2A	14-4
M2Settings	14-5
M2Errors	14-9
M2Prof	14-11
MakeErrorList	14-14
IFF2Obj	14-16
M2Batch	14-18
M2XRef	14-20
M2Debug	14-21

V. Easy Modules

15. EasyBeeper	
Using EasyBeeper	15-2
Variables	
nextVolume	15-4
nextPeriod	15-5
nextCycles	15-6
Procedures	
Beep	15-7
16. EasyDBuf	
Using EasyDBuf	16-3
Variables	
nextDetailPen	16-4
nextBlockPen	16-5
nextModes	16-6
nextFlags	16-7

nextTextAttr	16-8
Procedures	
CreateDBufScreen	16-9
CloseDBufScreen	16-10
SwapDBufScreen	16-11
SetDBufScreenColor	16-12
17. EasyGadgets	
UsingEasyGadgets	17-2
Variables	
listFailed	17-4
currentList	17-5
currentGadget	17-6
currentStrBuffer	17-7
currentIntBuffer	17-8
currentIntui	17-9
nextBoolActivation	17-10
nextIntActivation	17-11
nextPropAcgtivation	17-12
nextStrActivation	17-13
nextIsReqGadget	17-14
nextFlags	17-15
nextID	17-16
nextIntuiTextAttr	17-17
nextIntuiFrontPen	17-18
nextIntuiBackPen	17-19
nextIntuiDrawMode	17-20
nextIntuiLeftEdge	17-21
nextIntuiTopEdge	17-22
nextBorderFrontPen	17-23
nextBorderDrawMode	17-24
nextShadowFrontPen	17-25
Procedures	
StartList	17-26
DisposeList	17-27
AddBoolGadget	17-28
AddBoolImageGadget	17-29
AddIntGadget	17-30
AddPropGadget	17-31
AddStrGadget	17-32
AddBorders	17-33
AddDropShadow	17-34

18. EasyGamePort	
Using EasyGamePort	18-2
Variables	
joyFailed	18-3
Procedures	
ReadJoy	18-4
WaitJoy	18-5
 19. EasyGels	
UsingEasyGels	19-3
Variables	
nextVFlags	19-5
nextBFlags	19-6
Procedures	
CreateBob	19-7
DisposeBob	19-8
CreateGelsInfo	19-9
DisposeGelsInfo	19-10
CreateVSprite	19-11
DisposeVSprite	19-12
CreateBuffer	19-13
DisposeBuffer	19-14
 20. EasyIDCMP	
Using EasyIDCMP	20-2
Procedures	
ProcessEvents	20-4
 21. EasyMenus	
Using EasyMenus	21-2
Variables	
stripFailed	21-3
currentStrip	21-4
currentMenu	21-5
currentItem	21-6
currentSub	21-7
currentIntui	21-8
nextMenuLeftEdge	21-9
nextItemLeftEdge	21-10
nextSubLeftEdge	21-11
nextIntuiLeftEdge	21-12
nextItemTopEdge	21-13

nextSubTopEdge	21-14
nextIntuiTopEdge	21-15
nextItemWidth	21-16
nextSubWidth	21-17
nextItemHeight	21-18
nextSubHeight	21-19
nextMenuFlags	21-20
nextItemFlags	21-21
nextSubFlags	21-22
nextTextAttr	21-23
nextFrontPen	21-24
nextBackPen	21-25
nextDrawMode	21-26
Procedures	
StartStrip	21-27
DisposeStrip	21-28
AddMenu	21-29
AddItem	21-30
AddSub	21-31
22. EasyPrintPict	
UsingEasyPrintPict	22-2
Procedures	
PrintRastPort	22-3
23. EasyProps	
Procedures	
SetProp	23-3
GetTopLine	23-4
24. EasyReadPict	
UsingEasyReadPict	24-2
Variables	
readState	24-3
currentFrame	24-4
Procedures	
ReadPicture	24-5
25. EasySavePict	
Using EasySavePict	25-2
Variables	
saveState	25-3

Procedures	
SavePicture	25-4
26. EasyScreens	
UsingEasyScreens	26-2
Variables	
nextDetailPen	26-3
nextBlockPen	26-4
nextModes	26-5
nextFlags	26-6
nextTextAttr	26-7
Procedures	
CreateScreen	26-8
SetscreenColors	26-9
27. EasySpeech	
Using EasySpeech	27-2
Variables	
speechFailed	27-3
currentPhoneme	27-4
nextVolume	27-5
nextSampFreq	27-6
nextRate	27-7
nextPitch	27-8
nextSex	27-9
nextMode	27-10
nextMouth	27-11
Procedures	
ReadMouth	27-12
Say	27-13
SayAndReturn	27-14
StopSpeech	27-15
WaitSpeech	27-16
28. EasyWindows	
UsingEasyWindows	28-2
Variables	
nextDetailPen	28-3
nextBlockPen	27-4
Procedures	
CreateWindow	27-5

VI. Appendix

- A. Error messages and return codes**
 - Compiler errors A-1
 - Return codes for support programs and utilities A-26
- B. Suggested reading**
- C. Glossary**
- D. Technical support**
 - Pre-call checklist D-1
 - Joining BIX for technical support D-3
- E. Customizing M2Data files**
 - M2Data contents E-2
- F. Run-time license agreement**
- G. M2E ARexx command summary**

VII. Index

1. M2Sprint and the Amiga

Welcome to M2Sprint, the first truly integrated programming environment for Commodore-Amiga computers. M2Sprint has been designed from the ground up to radically change the way programming is done on the Amiga.

M2Sprint is a giant step ahead of the many "user-hostile" language systems available today. Unlike some packages that force the user to create software "their way", M2Sprint offers a choice.

If they wish, programmers can take advantage of M2Sprint's highly advanced editor that incorporates state-of-the-art features to allow users to write, compile, link, and launch programs all from within a user-configurable environment.

Or, programs can be written with any Amiga text editor and be compiled and linked from the CLI, Workbench, or via the ARexx macro programming language.

Either way, users can benefit from M2Sprint's components that have been designed and optimized for the Amiga. The compiler and linker produce tight code very quickly, the Modula-2 oriented editor allows for integrated program development, and the straightforward library of modules provides easy access to all Amiga, Modula-2, ARP, and ARexx functions.

Whether M2Sprint is used for educational, recreational, or professional software development, it offers Amiga programmers the highest level of performance for the lowest cost.

M2Sprint components

- M2C - a high speed single-pass Modula-2 compiler
- M2L - a high speed single-pass Modula-2 program linker

- M2E - a multi-file Modula-2 text editor optimized for Modula-2 source code production
- M2Sprint support programs and utilities

M2Sprint library

- Amiga interface
- Standard Modula-2 functions
- Easy Amiga interface
- 'C' language style routines
- IFF (Interleave File Format) routines
- ARP (AmigaDOS Resource Project) interface
- ARexx macro language interface
- Utility modules

2. Reading the Programmer's Guide

The Programmer's Guide contains the technical and tutorial information needed to use all of M2Sprint's components. Its companion guide, the Library Module Reference, contains the listings for the definition modules included with M2Sprint.

Not all Amiga users have an equal level of understanding of their machines. For this reason, some sections of the Programmer's Guide are designed to bring novices up to speed, and others are dedicated to the discussion of some of M2Sprint's more advanced features.

Making the most of your time

The following list of suggestions is not meant as an attempt to classify users and somehow limit the use of the Programmer's Guide, but rather as a helpful attempt to optimize the time spent with it:

Novice Amiga users should:

- familiarize themselves with the basic operation of their machines by reading the "Introduction to the Amiga" manual included with each Amiga system
- read all sections of the Programmer's Guide in a linear (cover to cover) style and keep it nearby as a handy reference
- pay special attention to Chapter 8 - "Using the environment" in order to understand the basic role of each M2Sprint tool in the context of a step-by-step explanation on using the editor, compiler, linker and support utilities to create a simple program
- read Chapter 9 - "Library overview", an introduction to the nature of the library of modules provided with M2Sprint.
- refer to Appendix C - "Glossary" when unfamiliar terms or concepts are used

Intermediate users may wish to:

- read all of Section III - "Understanding M2Sprint".
Chapter 8 - "Using the environment" helps users understand how each component of M2Sprint can be used to speed program development.
Chapter 9 - "Library overview", an introduction to the nature of the library of modules provided with M2Sprint.
Chapter 10 - "Using the libraries" gives background information on how M2Sprint's libraries provide an interface with the Amiga's system routines.
- study all chapters of Section IV - "M2Sprint Components" in order to become aware of the capabilities of the editor, compiler, linker and various support programs

- explore and experiment with the demo programs (and source code) included on the Demos disk

Experienced users may benefit most by:

- skimming lightly over the bulk of the purely informational and tutorial material of each chapter
- reading the special **NOTE**'s inserted in the body of the text that draw attention to key features or important points
- paying special attention to Appendix E - "Customizing M2Data files" which explains how to modify M2Sprint's various data files in order to customize the working environment

The Programmer's Guide does not explore the Modula-2 language, or Amiga system software fundamentals. The assumption is made that the user has a basic understanding of Modula-2 and the Amiga or has access to the appropriate reference material. For a list of recommended reading materials consult Appendix B - "Suggested Reading".

Manual conventions

The Programmer's Guide uses certain conventions to help convey information consistently with minimum confusion.

Typography

- the **TIMES** font is used for the general body of text
- the **HELVETICA** font is used in headings and step-by-step instructions
- the **COURIER** font refers to text that should be typed in directly by the user. This font is also used throughout the manual to display program examples or sections of Modula-2 code
- bullets "•" are used to group and display lists of related program options or information

Commands

Keyboard commands and menu equivalents are referenced throughout the manual in the following manner:

Alt + x	press key x while holding the Right Amiga key
CTRL + x	press key x while holding the control key
SHIFT + x/y	press key x or y while holding the shift key

Menu commands are usually referenced throughout the manual in the following manner:

Project/Save refers to the "Save" item located in the "Project" menu

3. Contacting M2S

M2S' International and North American offices provide full sales, upgrade, and technical support for their respective regions.

Defective or damaged product

If your product is defective or damaged contact Customer Relations. Defective disks will be replaced free of charge during the 30-day Limited Warranty.

Updates and product information

If you have questions regarding the availability of updates, or would like general information on M2S products, please contact Customer Relations. They can answer your questions regarding product features and system requirements.

If you have moved to a new address please call or mail in notification to Customer Relations. By keeping your address current, you will always receive notification of updates, and new products.

To contact M2S, please write or call the nearest office.

USA and Canada

M2S
P.O. Box 550279
Dallas, TX 75335
Tel: (214) 340-5256
Fax: (214) 341-9104

International

M2S
P.O. Box 393
Bristol BS99 7WU
U.K.
(0) 272-425573

M2S also provides online technical support for M2Sprint and other products on BIX, the BYTE Information Exchange. Users with modems can dial with their computers, join BIX and with the command "join M2S" can link up with M2S personnel and other M2Sprint users.

M2S also provides online technical support in the AmigaVendor forum of the Compuserve service.

For more information on obtaining technical support from M2S and how to become a BIX member, refer to Appendix D - "Technical Support" in this manual.

4. System requirements

The following represents M2Sprint's minimum system requirements:

Computer

M2Sprint works equally well on base configurations of the Amiga 500, 1000 and 2000 computers.

Operating system

In order for M2Sprint to function, it requires version 1.2 or higher of Kickstart, and V1.3 of Workbench. M2Sprint also requires the "arp.library" file to be present in your libs: directory. The ARP library is supplied with M2Sprint.

Memory

M2Sprint requires a minimum of 512K RAM. Users with one megabyte of memory or more will benefit from significantly shorter compile and link times with the aid of the Amiga's speedy RAM disk. Extra memory also helps to improve the interaction between M2Sprint's various components.

Storage

One disk drive is required, however because M2Sprint often requires files to be loaded from data diskettes, an extra disk drive (or a hard drive) eliminates tiresome disk swapping that can occur.

NOTE: Users with the absolute minimum system configuration, 512K and one disk drive, should be aware that performance and usability of all M2Sprint components will be severely hampered. Expanding your Amiga by adding extra memory or a second disk drive (it is not essential to do both right away) will allow you to take advantage of M2Sprint's full potential.

5. Before you begin

In order to protect your software investment, there are several things that must be done before using M2Sprint. Please check the contents of the M2Sprint package to ensure that you have received a complete unit, backup the master disks so that if your working copies fail all is not lost, and fill out and mail the registration card.

Information on recent software modifications and additional documentation for M2Sprint can be found in various files located in the "READ_ME" drawer on the M2Sprint disk. After making backup copies of the masters, please take the time to examine these files so that you are aware of any late enhancements made to this release.

What's in the box?

Please check the contents of the M2Sprint package to ensure that all documentation, diskettes, and supplementary information material have been included and are in proper working condition.

M2Sprint package contents

- 2 manuals
 - Programmer's Guide
 - Library Module Reference
- a holder containing 6 disks
 - M2Sprint
 - Library
 - Fastload Library
 - Lib Sources 1
 - Lib Sources 2
 - Demos
- supplementary information material
 - Program License Agreement
 - Registration Card

If any of the above items are missing, please contact M2S for an immediate replacement.

Making backups

It is essential that working copies of the M2Sprint master diskettes be made immediately. The data stored on computer disks is susceptible to electrical and magnetic fields, and can also be damaged by physical contact with dust, fingerprints or moisture.

By creating backups, or working copies, the master diskettes can be kept safe from any harm that might otherwise come to them during day to day use.

Backups of the master diskettes can be made in one of four ways:

- from Workbench with two drives
- from Workbench with one drive
- from CLI with two drives
- from CLI with one drive

NOTE: Before following any of the backup procedures, please ensure that the master diskettes are write protected. If the write protect hole on the disk is closed, the disk is write enabled. If the write protect hole is open, the disk is write protected. Ensure that the hole is open on the master diskettes at all times.

Amiga 2000 users whose second disk drive is external should substitute DF2: wherever DF1: appears.

From Workbench with two drives:

1. **Insert an M2Sprint master diskette in the second disk drive (DF1:) and put a blank diskette in the first disk drive (DF0:).**

The icon for the blank disk may appear as "BAD" if this diskette has never been used before.

2. **Drag the master diskette's icon onto the blank disk's icon.**
3. **Follow Workbench's instructions to insert the System disk into the internal drive (DF0:).**

4. After a few seconds, follow Workbench's request for the blank diskette to be re-inserted into DF0:. This blank or "scratch" disk will become the working copy.
5. Remove both diskettes after Workbench announces the copy is complete. Wait until all disk activity is finished (both drive lights must be off).

Workbench automatically names the destination disk as the source disk's name with the prefix "copy of" added to the name. It is necessary to use "Workbench/Rename" to remove the "copy of" portion of the disk name.

6. Repeat steps 1-5 for each of the remaining master diskettes.

Store the master diskettes in a safe place away from direct sources of heat, and electrical or magnetic fields. NEVER use the masters for anything but creating working copies.

From Workbench with one drive:

1. Put a master diskette into the disk drive.
2. Select the master diskette's icon by clicking it once with the mouse.
3. Choose Workbench/Duplicate.
4. Follow Workbench's instructions to insert the System disk into the disk drive.
5. After a few seconds, follow Workbench's request for the destination diskette to be inserted. This blank or "scratch" disk will become the working copy.
6. Carefully follow the rest of the instructions given on the screen.
7. Remove the diskette after Workbench announces the copy is complete. Be sure to wait until all disk activity is finished (the drive light must be off).

Workbench automatically names the destination disk as the source disk's name with the prefix "copy of" added to the name. It is necessary to use "Workbench/Rename" to remove the "copy of" portion

of the disk name.

8. Repeat steps 1-7 for each of the remaining master diskettes.

Store the master diskettes in a safe place away from direct sources of heat, and electrical or magnetic fields. NEVER use the masters for anything but creating working copies.

From CLI with two drives:

- 1. At the CLI prompt type "DISKCOPY DF0: TO DF1:" and press RETURN.**
- 2. Follow the instructions to insert the source diskette (a master disk) into DF0: (drive number one), and a blank or "scratch" diskette (to become the working copy) into DF1: (drive number two).**
- 3. Press RETURN.**
- 4. Remove both diskettes after the copy is complete. Be sure to wait until all disk activity is finished (both drive lights must be off).**
- 5. Repeat steps 1-4 for each of the remaining master diskettes.**

Store the master diskettes in a safe place away from direct sources of heat, and electrical or magnetic fields. NEVER use the masters for anything but creating working copies.

From CLI with one drive:

- 1. At the CLI prompt type "DISKCOPY DF0: TO DF0:" and press RETURN.**
- 2. Follow the instructions to insert the source diskette (a master disk) into the disk drive.**
- 3. Press RETURN.**
- 4. When they appear, follow the instructions to insert the destination diskette (a blank or "scratch" diskette to become the working copy) into the disk drive.**

5. Press RETURN.
6. Remove the diskette after the copy is complete. Be sure to wait until all disk activity is finished (the drive light must be off).
7. Repeat steps 1-6 for each of the remaining master diskettes.

Store the master diskettes in a safe place away from direct sources of heat, and electrical or magnetic fields. NEVER use the masters for anything but creating working copies.

Warranty registration

M2S is committed to giving users the best possible after-sales support. However, we can only help you if we know who you are.

Before proceeding any further, locate the warranty registration card and fill in the necessary information. M2S must receive this card no later than 30 days after the purchase of M2Sprint.

6. Installing M2Sprint

This chapter explains how to use the M2Install program to set up a working M2Sprint programming environment.

Even though M2Sprint is an easy to use programming environment, the sheer number of disks that come with the package are enough to discourage even experienced programmers, let alone beginners.

For this reason, an easy to use installation program, M2Install, is provided on the M2Sprint main diskette that will copy all the necessary files from the M2Sprint disks onto your Amiga system to get you started.

NOTE: If you haven't made backup copies of the M2Sprint diskettes, do so at this time. Consult Chapter 5 - "Before you begin" for information on backing up your diskettes.

Purpose

The M2Install program will install all the necessary M2Sprint files into your normal Amiga environment to enable you to use M2Sprint. Many files contained on the M2Sprint disks will not be installed. For instance, the sources to all of the M2Sprint library are not copied, neither are the demos. Only the files necessary to run the environment are installed.

An important point to stress is that M2Install will try to integrate M2Sprint into your existing system setup. This relieves you from learning a whole new environment. M2Sprint will appear as an extension of your system's capabilities rather than a whole new system.

Using M2Install

M2Install was designed to be easy to use and to ensure that your environment is correctly set up. It should be used by beginners and advanced users. As the program progresses through the installation process, it informs you every step of the way, allows you to skip certain operations, or quit at any time.

Before running M2Install, you should decide where you wish to install M2Sprint. Your decision will depend on the amount of memory your system has and which peripherals are available.

If you have a hard drive, you will find that the performance of the M2Sprint environment is improved by installing it on the hard drive.

If you do not have a hard drive, you will have to install M2Sprint on floppy disks. To do this, you must prepare two diskettes by initializing them (using the Workbench's "Special/Initialize" menu item), and name them "M2Sprint_Tools" and "M2Sprint_Lib".

Finally, the last consideration before running M2Install is the amount of memory in your system. M2Install can prepare your system for memory-based operation, or disk-based operation. If you have two megabytes of memory or more, you will find that installing M2Sprint for memory-based operation will significantly improve performance.

Running M2Install

To start M2Install, double-click the M2Install icon in the main M2Sprint disk. You will be presented with the title screen. The program contains several different screens, one for every step of the installation process. Each screen explains the intent of the next installation step and allows you to perform the operation or skip directly to the next one. This allows experienced users to have a tight control over the installation process, thus allowing the installation program to still be used even on heavily customized environments.

At any time during program execution, you can click the window's close gadget to exit. If any errors are detected during the installation process, the program will print a descriptive error message, and abort.

ARP library

The M2Sprint environment relies heavily that the ARP (AmigaDOS Resource Project) library is in your system's LIBS: drawer. The LIBS: drawer is normally located on your system's Workbench disk. If M2Install senses that you do not have the library or do not have a recent enough version of it, it will present a screen offering you the possibility to install the ARP library in your system. Please note that if you skip this step, none of the M2Sprint utilities will function.

Math library

The M2Sprint compiler requires the `mathieeedoubbas.library` file to function properly. This file comes standard with the 1.3 AmigaDOS Workbench disk, so it may already be in your system. If M2Install senses that the library is not present, it will offer you the option to install it. If you do not have the library and do not install it, the compiler will not work.

Installation type

The next two screens allow you to specify on which type of system you wish to install M2Sprint. You may specify memory-based operation or disk-based, and can specify either a floppy system or a hard drive system.

If you opt for hard drive operation, first make sure that your hard drive has a minimum of two megabytes available. M2Install will prompt you for the drawer name where the entire environment should reside.

For example, you may wish to install the environment in a drawer called "M2Sprint" which is in the "Languages" drawer on DH0:. You'd enter the path such as:

```
DH0:Languages/M2Sprint
```

M2Install would then create the M2Sprint drawer and the rest of the installation process would affect this drawer only.

If you opt for a floppy based system, make sure you have two blank initialized diskettes with the names "M2Sprint_Tools" and

"M2Sprint_Lib". M2Install will request that you prepare the two aforementioned blank diskettes. When you have done so, you simply click on "Continue".

At this point, M2Install will start the installation process. The first thing to be installed are the tools. The tools are all the executables which compose the system (compiler, linker, editor, etc.). When you are ready, start the tools installation process.

The program will prompt you to insert the correct diskettes. Simply follow the prompts closely, and everything will be copied for you.

The same process will be repeated for the M2Data files, and the M2Sprint library. Again, simply follow the prompts.

M2Install must now provide files so that the operating system will recognize the presence of the environment. This involves writing a small file to your system's S: drawer (typically found on your Workbench disk), and making a slight change to your system's startup-sequence.

The change that M2Install wants to perform on your startup sequence is minimal. All that is needed is the statement "EXECUTE S:M2Sprint-startup" to be added at the front of the file. This will be done automatically, and can easily be removed at a later time if so desired. Removing this statement from your startup sequence and rebooting will cause the M2Sprint environment to become invisible to the operating system.

Once your startup sequence modified, you must reboot your Amiga in order for the environment to be recognized by your system. Once you have rebooted, you can simply click the M2E icon, or type "M2E" from any CLI to load the M2Sprint environment.

Advanced Users

Refer to the next chapter to learn what the installation program copies, and how you can perform the installation yourself.

7. Customizing the environment

As was presented in the Chapter 6, M2Install is a very simple and efficient way of installing M2Sprint on any Amiga system. If you have a heavily customized environment, or would like to perform the installation process yourself, this chapter will explain what M2Sprint needs in order to function properly.

Overview

The M2Sprint environment can be subdivided in the following groups:

- Tools & utilities
- M2Data support files
- Module library
- Source code to the module library
- Demonstration programs

M2Install limits its actions to the first three groups, it does not copy either the demos or the source code to the module library. The files composing these groups are not necessary for M2Sprint to function, they are provided for documentation and reference purposes.

Logical assignments

The M2Sprint environment relies on the presence of 3 logical assignments. Logical assignments are an AmigaDOS feature which allows the user to declare a directory as a logical device. The ASSIGN command is used to perform logical assignments.

The three assignments required by the environment are:

M2Sprint: The place where all the tools reside
M2Data: The place where all the support data files reside
M2: The place where most of the .sym and .lnk files reside

The M2Sprint: assignment

The environment uses M2Sprint: when trying to access tools. Specifically:

- when the editor attempts to load the compiler, it searches first for "M2C" and if not found then searches for "M2Sprint:M2C"
- when the editor attempts to load the linker, it searches first for "M2L" and if not found then searches for "M2Sprint:M2L"
- when the activator program, M2A, attempts to load the editor, it searches first for "M2E" and if not found then searches for "M2Sprint:M2E"
- when a program is set up to use the M2Debug debugger, upon program crash, the program will automatically try to load the debugger as "M2Debug" and if not found then will try to load "M2Sprint:M2Debug"

The M2Install utility copies every M2Sprint tool to the M2Sprint: directory. In some environments it may not be necessary to have all the tools in the M2Sprint: directory. In fact, the only programs that must be in M2Sprint: are M2C, M2L, M2E and M2Debug. All other tools and utilities can be placed anywhere within your programming environment. For example, you might want to copy them to your C: directory.

The M2Data: assignment

The M2Data: directory contains several files used by the M2Sprint environment to control various features. M2Data: contains the following files:

M2.casecorrect	Used by the editor for the Correct Case feature
M2.compconfig	Compiler configuration
M2.editconfig	Editor configuration
M2.errorlist	Descriptions of every Modula-2 error, used by the editor
M2.link	Icon image used by the compiler
M2.linkconfig	Linker configuration
M2.prog	Icon image used by the linker
M2.searchpath	Search path used by the compiler and linker
M2.wordcomplete	Used by the compiler for the Complete Word feature.

These files are discussed in detail in Appendix E - "Customizing M2Data files".

The only way the M2Sprint environment can recognize these support data files is if they are in M2Data:. None of these files are essential. They can in fact all be removed, and the M2Sprint environment will still function. For optimal use of the environment though, it is strongly recommended to keep these files in M2Data:.

Here's a brief description of the consequences of deleting any individual file:

M2.casecorrect

When the editor fails to load this file, M2E disables the Correct Case and Correct Word features for the entire editing session.

M2.compconfig

When the compiler fails to load this file, M2C assumes default values for its configuration settings.

M2.editconfig

When the editor fails to load this file, M2E assumes default values for its configuration settings.

M2.errorlist

When the editor fails to load this file, M2E does not display descriptive error messages when stepping through compilation errors. Only error numbers are displayed.

M2.link

If the compiler is asked to generate icons and can't find this file, no icon is created.

M2.linkconfig

When the linker fails to load this file, M2L assumes default values for its configuration settings.

M2.prog

If the linker is asked to generate icons and can't find this file, no icon is created.

M2.searchpath

When the compiler fails to load this file, only the current directory and the M2: directory are scanned for .lnk and .sym files.

M2.wordcomplete

When the editor fails to load this file, M2E disables the Complete Word feature for the entire editing session.

The M2: assignment

The M2Sprint compiler and linker both use M2: as a default directory where they can find .sym and .lnk files. Both programs will first try and find .sym and .lnk files by looking in the current directory. If the files are not found, then the programs try to look in the M2: directory. If the files are still not found, then an attempt is made to load the M2.searchpath file described above. M2.searchpath contains additional directories to scan.

If no M2: directory exist, the compiler will try to look in the current directory. If the file is not found, a file requester will be popped up asking the user to specify the correct location for the file. No attempt will be made to load the m2.searchpath file.

Customization suggestions

Probably the most useful customization tip is to install the M2Sprint library in a recoverable ram disk (such as Commodore's RAD:) and assign the M2: directory there. This makes for very fast compilation and link speeds, but it does require a fair amount of RAM. The M2FastLoad utility can be used to quickly move the .sym and .lnk files to the recoverable ram disk when first starting.

As mentioned above only M2C, M2L, M2E and M2Debug need to be at all times in the M2Sprint: directory. All other tools can be copied anywhere within your environment. Certain utilities are very seldom used such as MakeErrorList and M2Batch. For such programs, you may economize disk space by not copying them to your environment. If you need them, they can always be found on your backup of the M2Sprint master diskette.

After using the environment for a while, you may find that there are certain library modules which your programs never use. You may delete these modules from your M2: directory if you require more disk space. It is recommended to only delete the .sym files and leave the .lnk files intact. This is because even though you may not use specific modules directly, some modules you use do rely on them. In such a case, M2L must have access to the .lnk files when linking.

Resident mode commands

A special feature of the 1.3 Amiga Shell is its ability to load commands as resident. All of the M2Sprint tools can be loaded as resident in Amiga Shell (other shells may not allow them to be loaded). All of M2Sprint's support programs and utilities are re-executable, but not reentrant. This means that when the tools are loaded as resident, only one copy of each program can be running at any one time.

NOTE: See Commodore's 1.3 Enhancer manual for more information on the difference between re-executable and reentrant programs.

8. Using the environment

Each component of the M2Sprint system plays a key role in the process of software development. The high speed compiler and linker, optimized libraries, useful tools, and integrated editing environment all perform specific tasks that make your time more productive.

This chapter helps to familiarize the novice and intermediate user with the M2Sprint components by stepping through the process of creating a simple program from start to finish.

NOTE: This chapter requires that M2Sprint is installed in your system and ready for use. If it is not, consult Chapter 6 - "Installing M2Sprint" for the correct procedures on how to do so.

Text editing with M2E

Because the majority of all programmers' time is spent writing and debugging code, M2Sprint's editor has been created specifically for the job of working with Modula-2 source code. M2E's speedy scrolling, intuitive interface, and advanced features make it the perfect tool for all your text editing needs.

Although you needn't employ M2E to write your code, there are major time saving features that make using it well worthwhile. One example is M2E's integration of M2Sprint's compiler, linker and error lister so that your programs can evolve from idea to executable without you ever having to leave the user configurable programming environment.

Starting M2E

M2E can be run from CLI or Workbench as a standard Amiga application. Try loading it now using one of the methods outlined below.

From Workbench:

1. **Locate the M2Errors program icon and double-click it.**

From CLI:

1. **Type "M2Sprint:M2E" at a CLI prompt**
2. **Press RETURN.**

If the editor has loaded successfully, a new M2E display window will have opened. If this has not happened after all disk access has finished, verify that M2Sprint has been installed correctly and there is enough memory available in your system and retry. If there is not enough memory, free some by quitting an existing application already in memory or reboot your system.

Using M2E

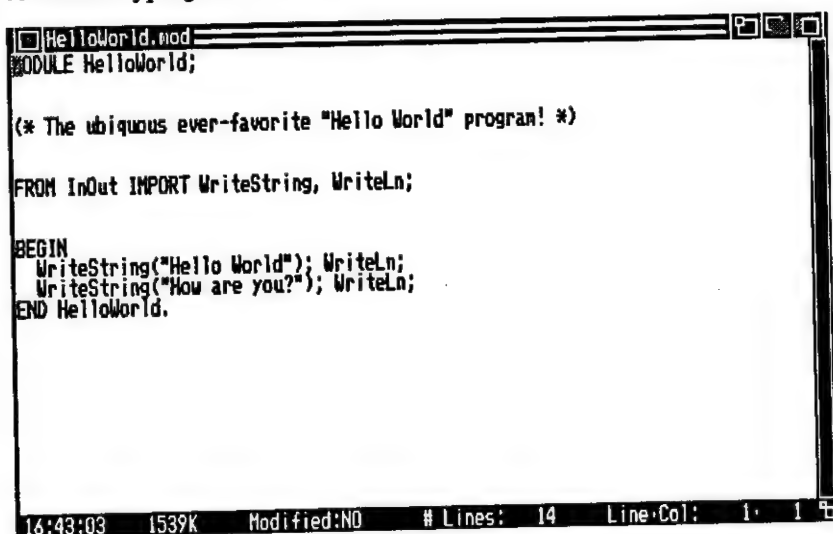
All text entry and editing takes place in the main text area of the M2E display window.

M2E fully supports the mouse and keyboard for text editing and cursor

movement functions. From the keyboard, the cursor can be moved up/down/left/right one position with the four directional arrow keys. With the mouse, the cursor can be moved to any position in the text area by pointing and clicking at the desired spot. The scroll bar can also be used to move the cursor vertically by dragging the filled portion of the scroll bar up or down. Clicking above or below the filled portion scrolls the display by one window.

New text is inserted at the cursor as it is typed.

Try practicing text entry and cursor control by entering the following short program example. The BACKSPACE and DEL keys can be used to correct typing mistakes.



```
MODULE HelloWorld;

(* The ubiquitous ever-favorite "Hello World" program! *)

FROM InOut IMPORT WriteString, WriteLn;

BEGIN
  WriteString("Hello World"); WriteLn;
  WriteString("How are you?"); WriteLn;
END HelloWorld.
```

NOTE: If "Config/Correct Case" is selected, Modula-2 keywords such as "FROM" and "BEGIN" are automatically converted to upper case after they are typed. Try typing Modula-2 keywords in lower case with and without this option on.

Verify that you have typed the file correctly.

Saving files

Select the "Project/Save" command to store your file to disk. Since

M2E requires a name to save the file under and your file is as yet untitled, the editor prompts you for a filename. Use "HelloWorld.MOD" for the filename. By default the file will be saved in the current directory. If this is not satisfactory enter a new pathname in the Drawer string gadget.

Click on "Save".

After saving, M2E returns to "ready mode" and you may continue editing. Note that the title bar now contains the name of the file.

If saving was not successful M2E's status bar will display the message "Error while saving!". Ensure that the correct drawer and filename were entered and that there is room on the disk.

NOTE: If you wish to have an icon generated for the file for use from Workbench make sure that "Config/Save Icons" is selected before the save.

Compiling with M2C

M2Sprint's compiler is a compact high speed single-pass compiler with a full implementation of the Modula-2 programming language. The compiler's job is to translate Modula-2 program and implementation modules into their MC68000 machine language equivalent code and definition modules into symbolic data.

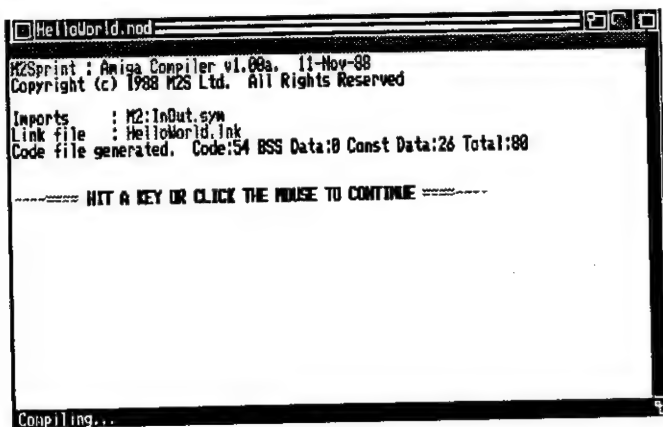
M2C accepts text files containing Modula-2 definition modules, implementation modules, or a list of filenames for batch compilation. M2C can be used from CLI, Workbench, or from inside M2E.

To compile the "HelloWorld" program, follow one of the methods as outlined below.

NOTE: Users with 512K memory may need to exit M2E and compile from CLI or Workbench in order to free the required memory.

To compile "HelloWorld" from inside M2E:

1. Select "Modula-2/Compile".



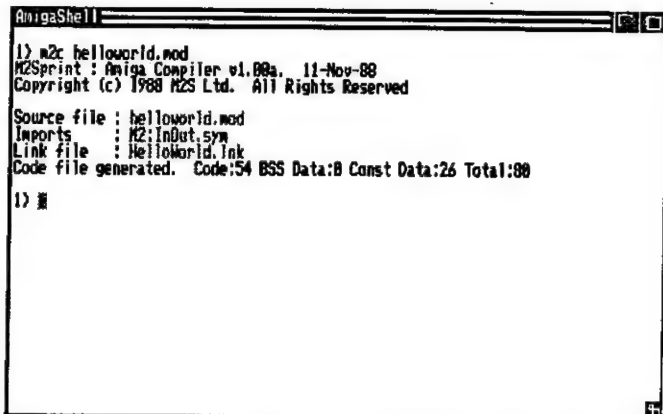
M2E loads the compiler from disk for the first compilation only and stores it in memory until the current session with M2E is finished. This means that loading time for the compiler in subsequent compilations is cut to nil. The memory used by the compiler may be freed by using the "Modula-2/Unload Comp/Link" command.

To compile "HelloWorld" from Workbench:

1. Locate and select the "HelloWorld.MOD" file icon by clicking it once.
2. Double-click on the M2C icon while holding down the SHIFT key.

To compile "HelloWorld" from CLI:

1. Make sure that the "HelloWorld.MOD" file is saved in the current directory.
2. Type "M2Sprint:M2C HelloWorld.MOD" at the CLI prompt.
3. Press RETURN.



```
AmigaShell
1) m2c helloworld.mod
M2Sprint : Amiga Compiler v1.00a, 11-Nov-88
Copyright (c) 1988 M2S Ltd. All Rights Reserved

Source file : helloworld.mod
Imports     : M2:InOut.sym
Link file   : HelloWorld.lnk
Code file generated. Code:54 BSS Data:8 Const Data:26 Total:80

1) █
```

After compilation, check to see if the files "HelloWorld.lnk" and "HelloWorld.ref" have been put in the current directory. Their presence indicates that the compilation was successful.

NOTE: If the file "HelloWorld.ref" cannot be found, check to see if the "Generate Reference File" option in the "Config/M2 Settings.." requester is on. The presence or absence of the reference file has no bearing on the outcome of this example compilation.

Icons will only appear after the Workbench window that they reside in is closed and reopened.

Finding bugs with M2Errors and M2E

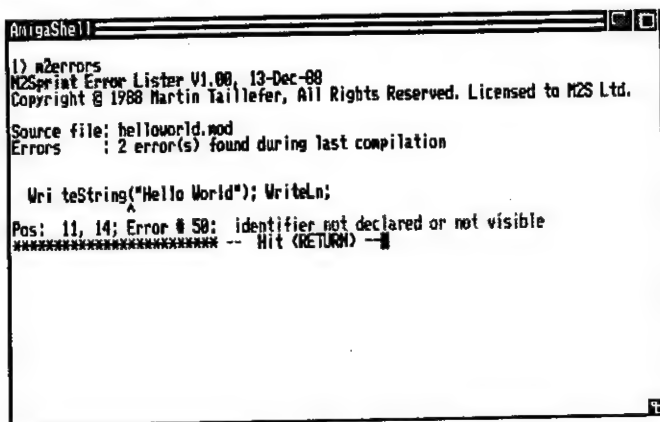
The M2Errors utility and M2E simplify the job of debugging source code. Errors found by M2C during the last compilation, can be viewed with the special M2Errors utility or with the integrated editor M2E.

M2Errors interprets the "T:M2.errorlog" file generated from the most recent compilation. It displays each mistake with the offending source code, line and column position of the error, error number, and the corresponding "plain english" description for each error.

M2E has an advanced error displaying feature built right into the program. After a file has been compiled from inside M2E, the entire list of errors can be stepped through one at a time using M2E's "Module-2/Next Error" command. This easy to use function positions the cursor at the error for correction and displays the error code and description in the status bar.

To find errors using M2Errors from CLI:

1. Type "M2Sprint:M2Errors" at the CLI prompt.
2. Press RETURN.



```
AnigaShell
1) m2errors
M2Sprint Error Lister V1.00, 13-Dec-88
Copyright © 1988 Martin Taillefer, All Rights Reserved. Licensed to M2S Ltd.

Source file: helloworld.mod
Errors      : 2 error(s) found during last compilation

  Wri teString("Hello World"); WriteLn;
Pos: 11, 14; Error # 58: identifier not declared or not visible
***** -- Hit (RETURN) --
```

M2Errors lists each error (if any) and waits for the user to press RETURN before continuing to the next one.

To find errors using M2Errors from Workbench:

- 1. Locate the M2Errors program icon and double-click it.**

M2Errors lists each error (if any) and waits for the user to press RETURN before continuing to the next one.

To find errors using M2E:

- 1. Select "Modula-2/Next Error".**

This moves the cursor to the point of the first error and displays the related error information in the status bar.

- 2. Step through each mistake with "Modula-2/Next Error".**

When the last error has been reached, choosing "Modula-2/Next Error" once more returns to the first error at the top of the list.

Correct any errors that may have been found by M2C and recompile the "HelloWorld.MOD" program.

If errors persist, verify the file against the source code listed earlier.

Linking with M2L

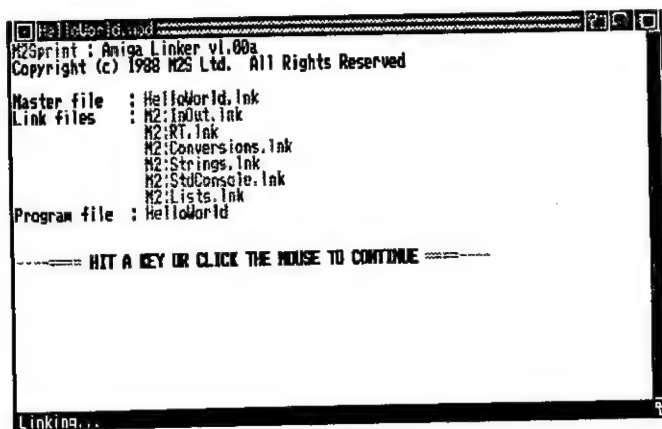
After all errors have been eliminated and a successful compilation has been done, the linker must be used to generate the final executable. M2L's job is to link together the compiled modules that are used by the main program and IMPORTED modules.

To link the "HelloWorld" program, follow one of the methods as outlined below.

NOTE: Users with 512K memory may need to exit M2E and link from CLI or Workbench in order to free the required memory.

To link "HelloWorld" from Inside M2E:

1. Select "Modula-2/Link".
2. Enter "HelloWorld" in the "Module Name?" requester that appears.
3. Press RETURN.



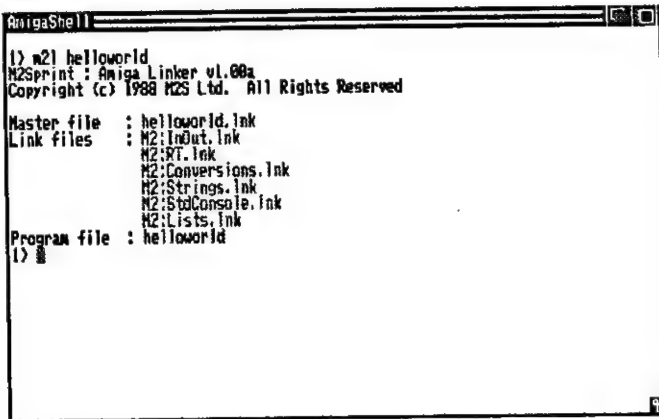
As with the compiler, M2E loads the linker for the first link only and stores it in memory until the current session with M2E is finished. The linker may be cleared from memory by using the "Modula-2/Unload Comp/Link" function.

To link "HelloWorld" from Workbench:

1. Double-click on the M2L icon.
2. At the "Link >" prompt enter "HelloWorld".
3. Press RETURN.

To link "HelloWorld" from CLI:

1. Make sure that the "HelloWorld.lnk" file is present in the current directory.
2. Type "M2Sprint:M2L HelloWorld.MOD" at the CLI prompt.
3. Press RETURN.



```
AmigaShell
l> m2l helloworld
M2Sprint : Amiga Linker v1.00a
Copyright (c) 1988 M2S Ltd. All Rights Reserved

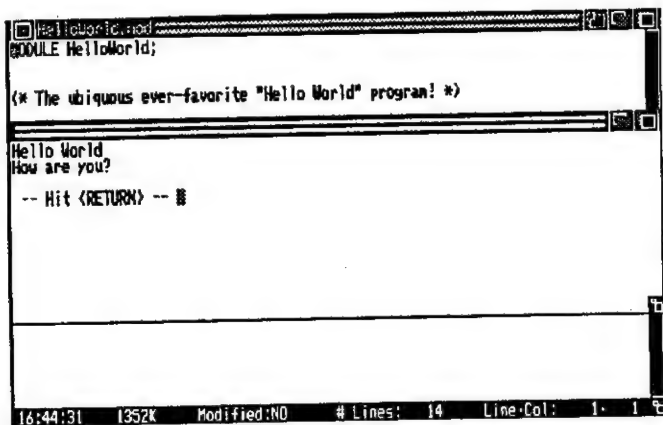
Master file : helloworld.lnk
Link files  : M2:InOut.lnk
              M2:RT.lnk
              M2:Conversions.lnk
              M2:Strings.lnk
              M2:StdConsole.lnk
              M2:Lists.lnk
Program file : helloworld
l> █
```

Running "HelloWorld"

If linking was successful, an executable file called "HelloWorld" will have been generated and saved in the current directory. If it's there, try running it.

Running "HelloWorld" from Inside M2E:

1. Select "Modula-2/Run Program".



```

HelloWorld.mod
MODULE HelloWorld;

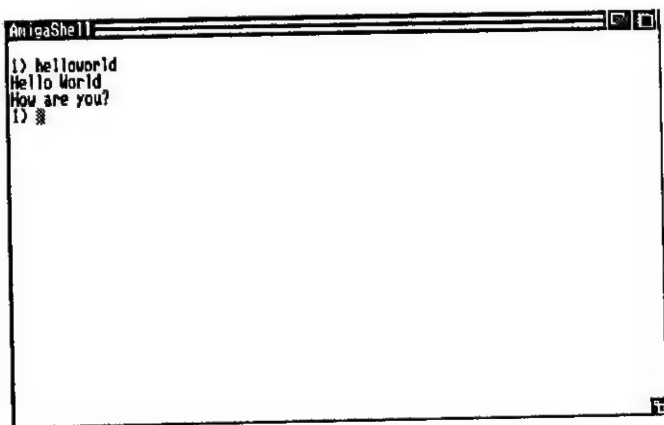
(* The ubiquitous ever-favorite "Hello World" program! *)

Hello World
How are you?
-- Hit <RETURN> --

16:44:31 1352K Modified:NO # Lines: 14 Line-Col: 1 1
```

Running "HelloWorld" from CLI:

1. Type "HelloWorld".



```

AmigaShell
1) helloworld
Hello World
How are you?
1) 
```

Running "HelloWorld" from Workbench:

- 1. Double-click on the "HelloWorld" icon.**

9. Library overview

M2Sprint's unparalleled library of over 160 modules provides novice, intermediate, and expert programmers with a massive resource of functions that cover nearly every aspect of the Commodore-Amiga computer.

Every element of each module has been tried and tested to ensure that they will perform without error when used in your programs. And because the source code to every module has been included with your M2Sprint package, modifications can be made to suit your personal requirements without difficulty.

The library of modules has been divided up into different sections, each covering a certain field of material:

- Amiga interface
- Standard Modula-2 functions
- Easy Amiga interface
- 'C' language style routines
- IFF (Interchange File Format) interface
- ARP (AmigaDOS Resource Project) interface
- ARexx macro language interface
- Amiga utility modules

Please note that this chapter gives only an overview of each section of the library. For specific information on the functions contained in a specific module, refer to the Library Module Reference.

Amiga system interface

The Amiga system interface is the largest of any single section of the library accounting for almost half the total number of modules provided. These modules provide an interface to the machine's native ROM-based and disk-based system libraries.

Experienced programmers who are already familiar with the Amiga's operating system should have no trouble using these modules as each procedure and variable follows its Commodore 'C' counterpart closely.

Programmers who have had little or no experience in using Amiga system routines may find their first attempts to do so quite frustrating. With an operating system as complex as the Amiga's, the "caveman" trial-and-error approach to programming simply will not work. Many would-be Amiga programmers have become quickly disillusioned with their computer by trying to write complex programs using windows, menus, and gadgets without the benefit of proper reference material and examples.

Investing in one or more of the Amiga books listed in Appendix B - "Suggested reading" and spending some time closely examining the example programs supplied on the Demos disk will help you avoid many of the most common programming mistakes that can quickly lead to frustration.

Standard Modula-2 functions

Like every other implementation of Modula-2, M2Sprint includes a set of standard modules to round out Modula-2's capabilities. Although there are no absolute standards for these modules, there are guidelines detailing the facilities that should be provided. M2Sprint has adhered to these recommendations and as such very few differences exist between M2Sprint's modules and those that exist for other systems.

The standard Modula-2 modules provide a fairly easy to use set of commands that cover basic I/O, memory management, type conversion, and mathematical functions. The advantage of writing programs using elements from this group of modules is that they can be easily transported and recompiled with any Modula-2 system on any computer.

The disadvantage is that the standard modules are kept as generic as possible in design to maximize the level of compatibility between systems. As such the procedures in these modules provide no control over Amiga specific system primitives such as color, sound, graphics, and interface conventions like menus, windows, mice, and gadgets.

If your programs need to be compatible with other Modula-2 systems, or are extremely simple in design then use the provided procedures. However to access the Amiga's higher functions, it will be necessary to use procedures from the Amiga section of M2Sprint's library.

Easy Amiga Interface

The Amiga is a complex system with a steep learning curve. To learn how to set up a program in order to simply open and manage a single window can require hundreds of pages of reading and many hours of trial and error programming. The purpose of the M2Sprint Easy modules is to allow quick turn-around time in developing programs that use Amiga specific functions. All the grunt work associated with initializing the dozens of records needed to program the Amiga is done automatically, and all that is needed is to call simple, easy-to-use procedures.

Using the M2Sprint Easy modules will tremendously simplify your Amiga programming. It will make your programs a lot easier to write and maintain. These modules will increase your productivity, and will reduce the number of bugs that can appear in your programs as you develop them. Since the modules work and are proven, you do not have to worry about their validity (and all the underlying records they handle), and can concentrate instead on your own code.

Although the Easy modules are simple to use they still provide full control over every detail that they are manipulate.

Module overview

Each Easy module is made up of variables and procedures which control one specific feature of the Amiga software system. The modules are:

- **EasyBeeper** - Provides a single procedure that controls the Amiga audio device to produce beeps of various tones and intensities. This is useful to send warnings or attention signals to the user.
- **EasyDBuf** - Provides an Intuition-compatible scheme to perform very simple double-buffering. Double-buffering is a technique used to produce smooth animation.
- **EasyGadgets** - Provides several procedures to easily create attractive and functional displays using the standard Amiga gadgets.

- **EasyGamePort** - Provides procedures to read data originating from joysticks plugged into the Amiga's game ports.
- **EasyGels** - Provides procedures to create and manipulate Amiga GELS (Graphics ELementS) used for animation. These include BOBS (Blitter OBjectS) and VSprites (Virtual Sprites).
- **EasyIDCMP** - Provides a single procedure that greatly simplifies the handling of Intuition messages sent to a window via that window's IDCMP.
- **EasyMenus** - Provides several procedures to easily create attractive and functional menu layouts using the standard Amiga menu facility.
- **EasyPrintPict** - Provides a single procedure to dump any Amiga bit map to a printer.
- **EasyProps** - Provides procedures to manipulate Amiga proportional gadgets.
- **EasyReadPict** - Provides a single procedure to read in an IFF picture from a disk file.
- **EasySavePict** - Provides a single procedure to save an Amiga bit map as an IFF picture on disk.
- **EasyScreens** - Provides procedures to create and maintain Amiga screens.
- **EasySpeech** - Provides procedures to control the Amiga's narrator device and translator library in order to easily generate speech.
- **EasyWindows** - Provides procedures to create and maintain Amiga windows.

General interface specification

Generally, an Easy module consists of three major components:

- **"nextXXX" variables** - These variables contain values which affect the behavior of the procedures from the same module. All such variables are initialized to logical default values. You can alter these values at any time to modify the behavior of the module's procedures. An example of this is the "nextVolume" variable from the EasyBeeper module. The value of this variable controls the volume of the beep the next time the Beep() procedure is called. The variable has an initial value of 64 which represents the maximum volume.
- **"currentXXX" variables** - These variables are pointers to various objects created by a previous procedure call. For example, after adding a menu item to a menu strip using the AddItem() procedure from the EasyMenus module, you can get the address of the actual MenuItem record allocated by the call. This allows you to directly modify certain fields of the record for some special case that the EasyMenus module doesn't handle.
- **Procedures** - These provide the actual functionality of the modules. Their behavior is mainly determined by the values of the "nextXXX" variables.

'C' language support.

Programmers who wish to use 'C' language style functions in their own programs or are faced with the task of converting large chunks of 'C' code to Modula-2 may want to take advantage of the modules in this section of M2Sprint's library.

The 'C' language support modules contain routines that conform closely to those contained in the standard 'C' language libraries.

IFF interface

The Amiga's standard Interchange File Format is the accepted standard for storing graphic images, audio samples, and some word processing files.

These modules provide procedures to handle all your IFF I/O requirements and are based on the standard Commodore supplied 'C' routines.

ARP interface

This section of the M2Sprint library provides the Modula-2 bindings so that functions in the `arp.library` can be accessed. The `arp.library` file has been included on the M2Sprint disk.

ARP, the AmigaDOS Resource Project, provides a library of procedures covering a wide variety of areas such as memory management, pattern matching, date handling, and even a standard file requester.

Using ARP functions gives your programs added stability and reduced size because the procedures have all been thoroughly tested and are written entirely in assembly language.

The ARP library is the result of the collective effort of dozens of top Amiga programmers whose work was coordinated by Charlie Heath of MicroSmiths Inc. They have succeeded in their goal to make a library of standard functions generally available to Amiga programmers and users as an alternative to AmigaDOS' BCPL functions.

ARP is public domain and as such the `arp.library` provided on this release of M2Sprint can be freely redistributed.

For more information on ARP please contact:

MicroSmiths Inc.
P.O. Box 561
Cambridge, MA 02140
(617) 354-1224

ARexx macro language interface

Modules in this section of the library provide Modula-2 links to the ARexx macro programming language.

ARexx is a high-level interpreted macro language that is rapidly gaining acceptance in the Amiga community as a dominant force in the Amiga's multitasking environment. Software that has been designed to recognize ARexx commands can be linked together to permit applications to communicate together.

For example, since M2Sprint's compiler is ARexx compatible, it can be interfaced with any other ARexx compatible text editor allowing programs to be compiled from within that editor.

Also, since M2E also has a set of ARexx commands that provide full control over all editor functions, it too can be interfaced with other programs.

Note that the ARexx modules that M2Sprint provides simply allow programmers to develop software that is ARexx compatible. Writing code for and using the macro language requires the ARexx interpreter not included with M2Sprint.

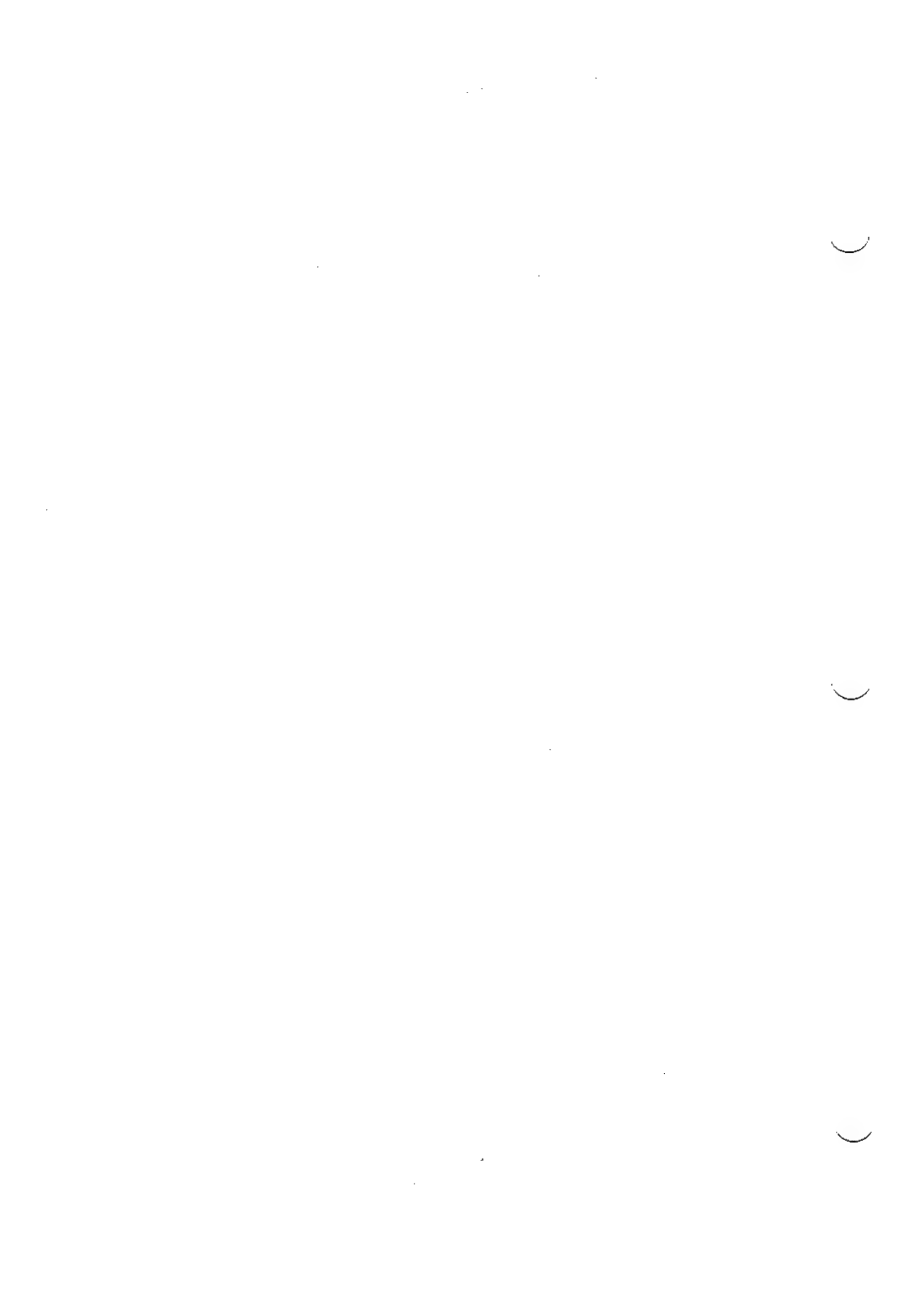
For more information on the ARexx programming language and interpreter, please contact:

ARexx
William S. Hawes
P.O. Box 308
Maynard, MA 01754
(617) 568-8695

Amiga utility modules

These modules provide routines identical in nature and functionality to the Commodore supplied utility include file for 'C' language compilers.

Each module provides utilities that can be used to control various specific elements of the Amiga's OS including Ports and Tasks.



10. Using the libraries

This chapter gives some guidelines on how to use the M2Sprint library of support modules to build solid and reliable Amiga applications. It does not pretend to teach you anything about the generalities of programming the Amiga, but instead focuses on how you can apply the information presented in the ROM Kernel Manual (and other reference material) to programming the Amiga using the M2Sprint environment. While reading this chapter, you may wish to have the Library Module Reference manual handy as well as the ROM Kernel and Intuition manuals.

Many of the concepts explored in this chapter have been implemented in the demonstration programs that came with M2Sprint. Reading the source code to these demos is one of the best ways to get acquainted with Modula-2 on the Amiga.

Overview

The Amiga Operating System (OS) is constituted from a complex hierarchy of interacting components. One of the original designs goals of the system was to keep it open-ended. All system structures had to be dynamic, and no fixed-sized data tables could be used. A result of these requirements is that the system architecture is completely address independent. There exists only one arbitrary fixed address in the Amiga system: location 4. Location 4 contains a pointer to the ExecBase data structure (see ExecBase.def) which is the anchor of the entire system, everything has some link to ExecBase.

Modula-2 and C

The Amiga ROM Kernel manual was written with C programmers in mind. All examples, and explanations assume that the reader is comfortable with the C language. To program the Amiga in Modula-2, it is essential to be able to perform conversions from C to M2. As an aid, many of the examples presented in the RKM have been converted to Modula-2 and included on the Demos disk. Here are some details useful to keep in mind when inspecting C code:

- The C NULL pointer is equivalent to the Modula-2 NIL constant.
- Many system routines require a pointer to specific data structures. To obtain the storage address of a structure, use the ADR() procedure from the SYSTEM module.
- The C bit fields have been implemented in M2Sprint using the SET concept. This works somewhat better and allows type-checking to take effect.

Exec

Exec is the real-time multitasking kernel which is at the heart of the Amiga system. It manages both CPU and memory resources, and provides arbitration protocols that allow other system components to manage their own respective resources.

As mentioned above, Exec is extremely dynamic. This fundamental characteristic has influenced the very essence of all other Amiga system components, and many of the third party applications developed to run on the Amiga. The main advantage of a dynamic system is that there are no size restrictions on anything; there are no fixed-size tables that dictate the maximum number of tasks in the system, no maximum amount of memory an application can support, etc. Everything is open-ended, expandable and very flexible.

To support this dynamic operation, the Amiga's operating system relies heavily on records. Most high-level system records are composed of lower-level records. This building-block approach makes for a consistent, easy to maintain system which encourages good programming style.

To maintain its dynamic nature, Exec uses doubly-linked lists. A standard type of linked-list is used, known as Exec lists. These lists are composed of a header known as a List record (Lists.def), and node elements known as Node (Nodes.def). Exec provides a standard set of routines to operate on these lists (Lists.def). When dealing with system lists, you will need to become familiar with these routines.

TYPE

```
(* normal node *)
NodePtr = POINTER TO Node;
Node = RECORD
    lnSucc : NodePtr;    (* the next node on the list      *)
    lnPred : NodePtr;    (* the previous node on the list  *)
    lnType : NodeType;   (* type of node this is          *)
    lnPri : BYTE;        (* [-128..127] the node priority *)
    lnName : STRPTR;     (* the nodes name, NIL means no nam *)
END;
```

TYPE

```
(* normal, full featured list *)
ListPtr = POINTER TO List;
List = RECORD
    lbHead : NodePtr;    (* head of the list      *)
```



```

    lhTail : NodePtr;      (* tail of the list      *)
    lhTailPred : NodePtr;  (* predecessor or tail *)
    lhType : NodeType;     (* type of list      *)
    lhPad : BYTE;
END;

```

As an example of the building block nature of the Amiga OS, consider the Message structure from Ports.def:

```

Message = RECORD
    mnNode : Node;          (* standard linkage component *)
    mnReplyPort : MsgPortPtr; (* message reply port      *)
    mnLength : CARDINAL;    (* message length in bytes  *)
END;

```

Note the Node structure at the head of the Message. This allows Exec to use the standard List routines to manipulate Message structures. This means Messages can easily be kept in lists or queues. This same approach is used with most other Exec-defined data structures. It is also very useful at the application level for general-purpose data management.

Exec has simpler versions of the List and Node structures known as MinList and MinNode. These structures involve less overhead than their full blown counterparts, but offer less flexibility. All the normal List functions can be applied to MinLists and MinNodes except for FindName() and EnQueue(). Additionally, no system type-checking on the node types can be performed.

```

TYPE
    (* stripped node - no type checking is possible *)
    MinNodePtr = POINTER TO MinNode;
    MinNode = RECORD
        mlnSucc : MinNodePtr;
        mlnPred : MinNodePtr;
    END;

```

```

TYPE
    (* minimum list -- no type checking possible *)
    MinListPtr = POINTER TO MinList ;
    MinList = RECORD
        mlnHead : MinNodePtr;
        mlnTail : MinNodePtr;
        mlnTailPred : MinNodePtr;
    END;

```

The most common operation performed on a list is a scan, having the program step through all of its elements. Due to the special circular nature of Exec lists, scanning them is slightly more involved than scanning your everyday linked-list. As an example, the following loop will go through all the elements of a list:

```
node:=list.lhHead;
WHILE node # ADR(list.lhTail) DO
  WriteString(node^.lnName^); WriteLn;
  node:=node^.lnSucc;
END;
```

Modula-2 has strong type-checking unlike C, which is somewhat more lenient in the area. For this reason, when using Exec lists in your own applications, it may be more useful to construct your own version of the Node structure to use at the head of all your nodes. For example, having the following declaration:

```
TYPE
  DataPtr = POINTER TO Data;
  Data = RECORD
    dtNode : MinNode;
    dtStuff : CARDINAL;
  END;
```

would not allow you to reference an item such as:

```
VAR
  node      : Data;
  ptr,next  : DataPtr;
BEGIN
  ptr:=ADR(node);
  next:=ptr^.dtNode; (* type incompatible *)
```

To ease this restriction, it is simple to declare a MinNode look-alike and use that instead:

```
TYPE
  MyMinNode = RECORD
    mnSucc: DataPtr;
    mnPred: DataPtr;
  END;
```

Using this structure instead of a MinNode would allow the above code fragment to compile correctly. The main advantage to using your own version of the MinNode is that it then becomes possible to dereference

other nodes from the current node easily. Additionally, all standard Exec list functions will still work correctly.

Libraries

Exec provides the Library concept as a means to store related functions and data structures. Each major Amiga component has its own library. Programmers can build libraries to expand the system.

Before using functions in a library, the library must first be opened. This is done using the `OpenLibrary()` procedure such as:

```
DiskfontBase := OpenLibrary(ADR(DiskfontName), 33);
```

this statement will open the disk font library. Once a library has been opened, and the proper library base variable initialized (in this case it is `DiskfontBase`), then the functions and data structures for that library may be used. Any attempt to use the functions from the disk font library prior to opening the library would inevitably result in a system crash. Remember, once a library has been opened, it must also be closed at some later point. Once the library is closed, the routines it contains are inaccessible until the library is reopened.

All system calls are attached to a library. This poses the classic "chicken or the egg" style problem of how to access the initial `OpenLibrary()` call? The library containing the `OpenLibrary()` function has not yet been opened, so how can one access the routine?

Happily the answer is simple. The system contains one library which is permanently opened: `ExecBase`. The Exec library contains all the essential routines used by the other system libraries. It does not need to be opened, before its routines can be used.

The Amiga comes equipped with several standard libraries, some in ROM, and some on disk in the Workbench LIBS: drawer. To perform any useful task, all programs have to open at least a few libraries upon startup. `M2Sprint` relieves you from the burden of opening and closing all of the built-in ROM libraries. The task is performed in the `RunTime` module which is linked to every executable.

Disk-based libraries (like `DiskfontBase`) are not opened automatically by the startup code. It is your responsibility to ensure that they are properly opened and closed by your application. The reason why they are not opened by the `RunTime` module, is that since they are disk-

based, the system would need to load them in memory when your programs start. Since most programs do not require these disk-based libraries, this would needlessly consume memory (to store the opened libraries) and would require some time for disk access.

M2Sprint uses four Amiga libraries to perform all of its REAL and LONGREAL arithmetic. There are two libraries used for REAL (one for simple math such as add and subtract) and the other for transcendental operations such as cosine and sine).

There are two equivalent libraries used for LONGREALs. Only the basic REAL library is contained in ROM, the three others are on disk. This means that every time you wish to use LONGREAL, or use transcendental REAL operations, the libraries must be opened and loaded from disk. M2Sprint provides the RealSupport module to do this. By using the appropriate call, the library will be opened and then closed automatically on program termination.

Devices

Another powerful tool provided by Exec is the Device (Devices.def). A device is very similar to a library, but the interface conventions and purpose of devices is somewhat different than those of a library.

An Exec device usually arbitrates the access to some physical hardware device, or in some cases to a logical hardware device. For example, the serial device handles access to the serial port hardware. An example of a logical hardware device is the ramdrive device. This device simulates the presence of an extremely fast hard disk, but is in reality totally software-based. The illusion is such, that the internals of the ramdrive could be modified to actually use a hard drive, and the rest of the system could not tell the difference (aside from a slow down in response time).

As with the libraries, devices need to be opened before any of their features can be used:

```
error:=OpenDevice(ADR(SerialName),0,ADR(ioreq),LONGBITSET{});
```

Once a device is opened, it must eventually be closed by CloseDevice().

Devices generally do not provide any user-callable function calls. Instead a few Exec functions are used to communicate with the device (IO.def). The call to OpenDevice requires a special data area (in our case it's ioreq). OpenDevice initializes certain fields of this data area. Depending on which device you use, the name and definition of the data area changes. When calling the IO routines, you must pass a pointer to this data region to the function.

11. M2E - The Editor

M2E's user-interface and program features are an intuitive blend of Amiga tradition and innovation. In addition to serving as an editor with enough speed and power to write the largest Modula-2 programs, it also acts as the hub of the integrated M2Sprint programming environment.

From inside M2E, files can be loaded, compiled, linked and launched with only a few keystrokes. Advanced features such as Complete Word, Correct Case, and ARexx macros enhance programmer's abilities to write Modula-2 programs. After compiling, M2E simplifies the job of debugging code by positioning the cursor directly on the faulty source statements and displaying a helpful error message.

Perhaps most importantly, M2E can be adjusted to suit user's personal tastes. Many of the editor's features can be adjusted and saved so that M2E automatically configures itself to its user's specifications each time it is used.

M2E is a carefully crafted state of the art Modula-2 programming workshop with all the latest high-tech tools thoughtfully provided for you within reach.

Launching M2E

A new M2E window can be opened as a standard application from Workbench, any available CLI shell or from inside an M2E window.

From Workbench:

1. Find the M2E icon located in the root directory of the M2Sprint disk.
2. Double-click on the M2E icon.

From CLI:

1. Type "M2Sprint:M2E" at a CLI prompt
2. Press RETURN.

From Inside M2E:

1. Select "Project/New Window".

NOTE: The newly created window is a separate and distinct task from the window from which it was created. Either window can be closed or its parameters modified without affecting the other.

A note on memory

The Amiga's multitasking operating system allows users to have as many programs running as necessary. Unfortunately, as the number of applications in the system increases, the available system memory decreases. With some editors, it can become prohibitively expensive for most Amiga users when several files need to be worked with simultaneously.

Happily, M2E is written in such a way that only one copy of the editor is required in memory no matter how many times the user loads it from CLI or Workbench. Once M2E has been launched, it automatically detects and eliminates any duplicates of itself that may appear in memory. This drastically reduces the overhead normally required to edit multiple files.

Hot key control

An important feature of M2E is the ability to use it for spot, or impulse editing. In a programming environment it can be awkward, or even impossible to access CLI and Workbench for loading purposes.

Activating the optional HOT switch at launch time provides control over M2E's accessibility. This feature forces M2E to stay resident in memory even when no M2E windows are open and eliminates the need to reload the program from disk between editing sessions.

To start M2E with the HOT feature active:

- **from CLI type "M2Sprint:M2E HOT"**
- **from Workbench, adjust M2E's icon tooltypes to include the word HOT**

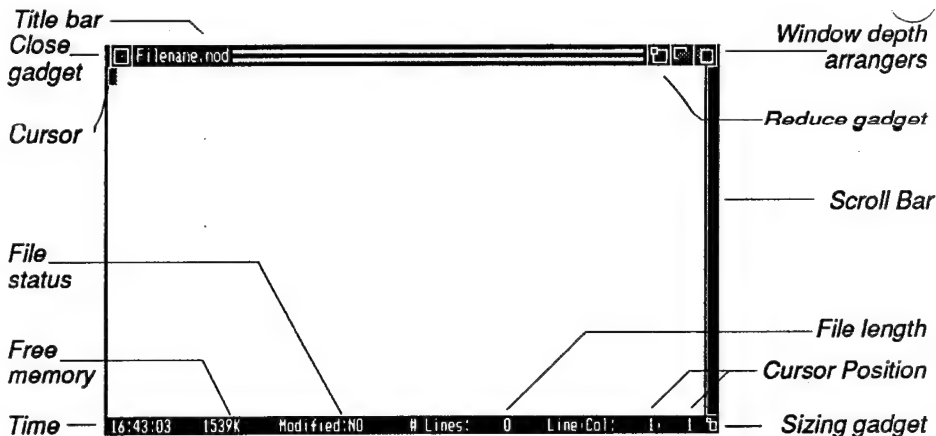
The HOT option allows users to open a new M2E window at any time by typing a special key combination CTRL + ALT + W. Typing CTRL + ALT + Y creates a new window and opens the file requester for loading. These two features can be used even if no M2E windows are open.

CTRL + ALT + Q turns the hot key option off and forces the program to remove itself from memory as soon as all M2E windows are closed.

M2Sprint's tiny M2A utility can be used from CLI or Workbench to quickly signal the editor to load files from disk into new windows.

The M2E display window

M2E's display window is the medium through which all text entry and editing takes place. The straightforward on screen layout provides the basic tools and file information for all text editing applications.



Title Bar

Displays the name of the file occupying that window.

Close Gadget

Closes the active window. User confirmation is required if modifications have been made since the file was last saved. Functionally equivalent to M2E's "Project/Close Window" command.

Window depth arrangers

Standard Amiga window-to-back and window-to-front gadgets. Clicking the right gadget brings the window to the front, clicking the left sends it to the back.

Reduce Gadget

A space saving device that toggles the window from regular size to the size of the title bar.

Cursor

Points to the current position in the M2E window. All text is inserted and deleted at the cursor position.

Sizing Gadget

Changes the width and height of the active window.

Scroll Bar

Allows the file in the current window to be scrolled.

The scroll bar consists of two parts; a vertical box representing the file's total length, and a solid bar inside the box representing the portion of that file being displayed in the window. A very small bar means that few lines are being displayed in the window in comparison with the total number of lines in the file. A bar that completely fills the box means that the entire file is being displayed.

Status Bar

Displays the system time, memory, file status, length, cursor position, as well as messages and warnings from M2E to the user. The status bar can be toggled on or off with the "Config/Status Bar" command.

All of M2E's requesters are movable and open at the position where they were last used. While requesters are up, files can still be edited by clicking in the main M2E window. Any number of requesters can be open at one time. Gadgets from the active requester can be selected from the keyboard by pressing the right Amiga key and the first letter of the gadget's name. Some examples follow:

A + Q Close any requester's window while it is active

A + B Toggles the "Backward" gadget in the Find requester.

File management

M2E provides the user with several basic file management functions that allow files to be created, stored, loaded, and printed quickly and easily.

Opening an existing file

Existing files can be opened in three ways:

- at a CLI prompt, by specifying the filenames for loading with M2E or the M2A utility
- from Workbench, by selecting the files' icons to be loaded and then double-clicking on M2E or the M2A utility
- from inside M2E with the "Project/New & Open..." or "Project/Open..." commands

Loading existing files from CLI:

Filenames may be passed to M2E for loading according to the following command line template:

M2E FILES/...

NOTE: AmigaDOS and ARP compatible wildcards may be included in any of the filenames.

Each file specified is loaded into its own individual window. If M2E is unable to match the specified filename with one on disk, it retries using the same name, but with the .MOD and .DEF extensions added. If M2E is still not successful in loading, a new window will be created using the original filename as default.

1. **Type "M2Sprint:M2E FileName" where FileName is the name of the file to be loaded. Multiple filenames may be specified by separating them with a space. Filenames containing spaces must be surrounded by quotation marks.**
2. **Press RETURN.**

Example

If M2E is started from CLI by typing "M2Sprint:M2E

MySource", the editor will try to load the file "MySource" from the current directory.

If the file "MySource" does not exist, M2E will automatically try to reload using the filename "MySource.MOD" and then if still unsuccessful, "MySource.DEF". If there is no match for all three filenames, a new window is opened with the name "MySource".

Loading existing files from Workbench:

Any single file with an M2E project icon can be loaded from Workbench by using the mouse to double-click on the file icon.

1. Find the file to be opened by inserting the appropriate disk and opening the necessary drawers.
2. Double-click the file's icon.

Multiple files can be selected for loading by shift-clicking their icons. Selecting Open from the Project menu in Workbench or shift-double-clicking M2E's application icon will launch M2E and load each file for editing.

1. Find the files to be opened.
2. Select the files to be loaded by shift-clicking their icons.
3. After all files are selected, double-click on the M2E or M2A icon while still holding down the SHIFT key.

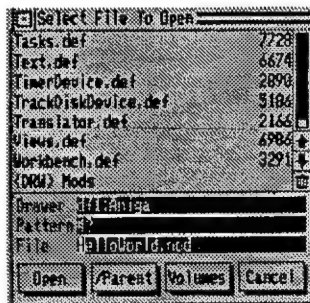
Loading existing files from within M2E:

1. Choose "Project/Open..." to load a file into the current M2E window or "Project/New & Open..." to create a window for the file to be loaded.

User confirmation is required with "Project/Open..." if changes have been made to the file since the file was last saved.

M2E's file requester lists all files and drawers in the current directory or disk that match the specified pattern.

The scroll bar and arrows may be



used to view the entire list. If the file to be loaded is not found, specify a new pathname by changing the Drawer name.

2. **Select the file by clicking its name from the list with the mouse, or by entering its name in the File string gadget.**
3. **Click "Open".**

Saving files

The importance of regularly saving your work cannot be overstated, especially in a programming environment. Should a system error or some other irregularity occur, it is likely that all modifications made to a file since it was last saved would be lost.

For this reason, M2E files can be saved to disk in two ways:

- manually with one of the save commands in the Project menu
- automatically with the Auto-Save feature

Storing files manually:

The Project menu contains three different commands to store the current file.

"Save"	Stores the file using the filename in the title bar
"Save As..."	Allows the user to specify a (new) filename before storing
"Save & Close"	Saves the file then closes its window

1. **Select one of the above commands from the Project menu.**

If the file is untitled, the file requester will open so that the new name and location of the file can be specified.

Storing files automatically:

M2E has the ability to automatically store a file at user specified intervals between saves. Saving will only occur if the file has been changed since its last save.

While "Config/Auto-Save" is selected, M2E will keep track of the elapsed time between saves. When this time interval exceeds the auto-save value, M2E will save the file using the current filename.

The delay between saves can be modified as follows:

1. Select "Project/Set Auto-Save Delay..."
2. Enter a number to specify how long (in minutes) M2E is to wait between saves.
3. Press RETURN.



Backups

As a protection against total loss of data due to a save error, M2E can be configured to backup the version of the file immediately before saving.

The file is preserved before each save as ":T/M2E.backup" if the "Config/Make Backups" option is selected.

NOTE: Backups are made only if a T directory exists on the same volume the file is to be saved on.

Printing

The "Project/Print" command sends the file in the active window to the system printer. Printing may be interrupted at any time by closing the special "Printing..." window that is open during printing.



Files are sent to the printer as defined by Amiga's Preferences. If irregular or unexpected output is being produced, check to make sure that the correct printer driver is selected for the attached hardware. Margins and character width settings should be adjusted from within Preferences if printing wide files.

Some printers are not able to handle TAB characters correctly. Additional formatting problems may result from TABs if the TAB width in M2E is set to a non-standard value (different than eight). "Extras/TABs to Spaces" can be used to convert TAB all characters to the equivalent number of spaces.

Cursor position and scrolling

The ability to move quickly and position the cursor easily are basic requirements of all text editors. M2E's intuitive operation and unparalleled scrolling speed eliminates much of the wasted time spent while editing files.

Positioning the cursor and scrolling can be done in three ways:

- from the keyboard with the cursor keys
- with the mouse
- with the scroll bar

Cursor position and scrolling from the keyboard:

M2E uses the keyboard's four directional (arrow or cursor) keys to adjust the position of the cursor. Hitting or holding down one of these keys moves the cursor one position in the direction indicated by the arrow. The SHIFT, CTRL, and ALT keys can be used in conjunction with the cursor keys to move by different amounts.

The following summary details the cursor movement key sequences and their functions:

Left/Right	Move cursor one character
Up/Down	Move cursor one line
SHIFT + Left/Right	Move by one word
SHIFT + Up/Down	Move by one window
CTRL + Left/Right	Move by 10 characters
CTRL + Up/Down	Move by 10 lines
ALT + Left/Right	Start/end of line
ALT + Up/Down	Start/end of file

M2E automatically scrolls the display if the cursor is moved outside of the limits of the window.

Positioning the cursor with the mouse:

1. Position the mouse pointer over the new location for the cursor.
2. Click the left mouse button.

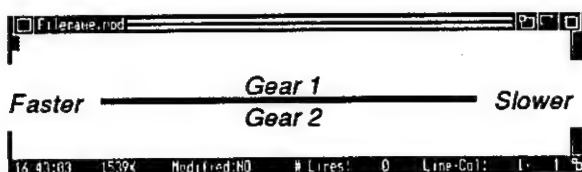
The cursor will follow the mouse pointer while the left mouse button is pressed and held down.

Scrolling with the mouse:

The mouse allows for variable speed control of text scrolling.

1. **Press and hold the left mouse button while the pointer is at the top/bottom of the window to scroll up/down.**
2. **Release the left mouse button to stop scrolling.**

The mouse pointer's position determines one of two "gears" that M2E scrolls with as well as the amount of "gas" to be applied.



To accelerate the scrolling speed in a particular gear, move the pointer to the left. To decrease the scrolling speed, move the pointer to the right.

Scrolling with the scroll bar:

Each M2E window has a vertical scroll bar that can be used to move in a file.

The scroll bar consists of two parts; a vertical box representing the file's total length, and a solid bar inside the box representing the portion of that file being displayed in the window. A very small bar means that few lines are being displayed in the window in comparison with the total number of lines in the file. A bar that completely fills the box means that the entire file fits in one window.

To scroll by one windowful of text, click on either side of the bar inside the box with the left mouse button. Clicking above the bar will shift the display up one window, and clicking below will shift the display down an equal distance.

Dragging the solid bar up or down causes M2E to shift the display to the appropriate spot in the file.

Editing

All of M2E's basic text functions are geared toward helping programmers create or modify files as quickly and easily as possible.

Files can be edited by selecting, cutting, copying, pasting, and deleting regions of text. New text can be entered in an M2E window by entering it directly with the keyboard, or by importing it from another file with the clipboard.

Selecting text

Selecting or highlighting text means choosing one or more characters in a given file. A variety of operations can be performed on selected blocks of text such as centering and justification. Most importantly however, selected text can be cut or copied from a file to the clipboard to be pasted inside the same file or exported to another.

Selecting text is a process of "marking" the start of the region to be selected, and then moving the cursor to the end of the block, thereby highlighting all the text in between the two points.

The highlighted region follows the cursor until the block has been cut, copied, pasted, or erased, or until the selection is cancelled.

To select a block of text:

1. Position the cursor at the beginning of the text to be selected.
2. Mark its position with the mouse by positioning the pointer over the cursor and clicking the right mouse button while holding down the left. This can also be accomplished with the "Edit/Mark" command.
3. Move the cursor to the end of the text to be selected.

Canceling a text selection

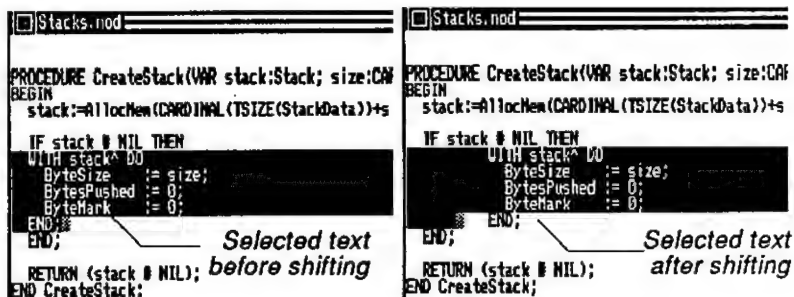
A highlighted region of text can be deselected and the selection mode turned off, by clicking the right mouse button or selecting "Edit/Mark Block".

Shifting selected text horizontally

A selected block of text can be shifted left or right for indentation or formatting purposes.

To shift text:

1. Select the text to be shifted.



2. Press CTRL + SHIFT + Left/Right to shift the text left or right.

Moving text

"Edit/Cut" and "Edit/Copy", are used to move selected blocks to a temporary storage area, called the clipboard. Once in the clipboard, the text can be inserted into any of the following places:

- the same file
- another M2E window
- any other Amiga application that supports the clipboard
- a new file

NOTE: Some programs do not use Amiga standard clipboard functions. For this reason transferring text to and from these applications via the clipboard may not work as documented with some programs.

Selected blocks can be moved from an M2E window to the clipboard in two ways:

- with the "Edit/Copy" command to duplicate the selected text and place it on the clipboard
- with the "Edit/Cut" command to remove the selected text from the window and place it on the clipboard

Cutting or copying text to the clipboard:

1. **Select the text to be moved.**
2. **Choose "Edit/Cut" or "Edit/Copy".**

Printing the clipboard

The "Edit/Print Clip" command is used when a portion of a file must be sent to the printer for hardcopy. This differs from the "Print" command as only the contents of the clipboard is printed instead of the entire file.

To print the contents of the clipboard:

1. **Highlight the text and copy it to the clipboard.**
2. **Select "Edit/Print Clip".**

Adding text to a file

New text can be placed in an M2E window from three sources.

- another M2E window or Amiga application via the clipboard
- an existing file
- the keyboard

Pasting text from another M2E window or Amiga application:

The "Edit/Paste" command is used to place text on the clipboard from another M2E window or Amiga application.

1. **Cut or copy the text to the clipboard from another application.**
2. **Position the cursor where the text is to be inserted.**
3. **Select "Edit/Paste".**

Pasting text from another file:

The "Edit/Open Clip..." function is used when it is necessary to insert the entire contents of an existing file into the current M2E window.

1. Choose "Edit/Open Clip..."
2. Select the file to be imported and click on Open (see the "Open..." command in the "M2E menu summary" section of this chapter for detailed instructions on how to use the file requester).
3. Position the cursor where the text is to be inserted.
4. Select "Edit/Paste".

NOTE: The "Edit/Open Clip..." function works best with files that contain plain ASCII text only. Specially formatted files (as those generated by most Amiga word processors) will not be read properly by M2E.

Saving the clipboard to a file:

1. Select the text to be moved.
2. Choose "Edit/Cut" or "Edit/Copy".
3. Select "Edit/Save Clip As..."
4. Use the file requester to choose a filename to save the clipboard under.

Typing new text

There are two basic typing modes in M2E, insert and overstrike.

In the insert mode, all text entered with the keyboard is inserted at the cursor's current position. After each character, the cursor shifts itself and all the text to the end of the line, one position to the right.

If "Config/Overstrike" is selected then the editor is in overstrike mode. Any text to the right of the cursor is overwritten with the new text, rather than shifted.

Auto-Indent

M2E's auto-indentation feature eliminates wasted time spent indenting each new line of code.

While "Config/Auto-Indent" is selected, new lines created by pressing the RETURN key will be started using the same indentation as the line above.

The ENTER key differs from RETURN in that it does not break the current line when used and always auto-indents regardless of the setting of "Config/Auto-Indent".

Word Wrap

Word wrapping is a feature often found in word processors, but seldom in text editors. Most text editors are designed for programming applications only, but with Word Wrap and other text formatting commands, M2E can be useful for simple word processing jobs as well.

If "Config/Word Wrap" item is not selected, a press of the RETURN or ENTER key must be issued in order to start each new line of text.

While "Config/Word Wrap" is selected, M2E automatically inserts a carriage return before the current word, when the cursor reaches the right margin. Although not generally used when writing programs, this can be very useful when writing documentation, commenting code, or creating other files that are made up of large paragraphs.

Configuration

One of the main advantages M2Sprint's integrated programming environment has over rival command-line compilers is the ability to configure its parameters to suit a particular application.

Many of M2E's parameters and options are user-definable. The Config and Extras menus are full of items that can be selected, disabled, or adjusted as needed. These items affect not only the editor's behavior, but also that of the compiler, linker, and the program being created.

M2E's "Config/Save Config" function allows the editor's default environmental settings and requester positions to be saved to the M2Data: drawer in a configuration file. M2E reads this file at the start of each session and sets up the editor according to the values contained within.

By default, a new M2E window will open to full screen size. This parameter may be adjusted by sizing and positioning the window to the desired dimensions, and selecting "Config/Save Config". All M2E windows opened from that point on will use the newly defined default window values. This feature is of particular importance to European Amiga users who may wish to adjust the default window size to match the PAL video standard's greater vertical display size.

The following is a list of adjustable control variables which are stored in the config file:

Extras Menu

- Set Auto-Save Delay...
- Set Right Margin...
- Set TAB Width...

Config Menu

- ARexx Console
- Auto-Indent
- Auto-Save
- Correct-Case
- Give Warnings
- Make Backups
- Overstrike
- Save Icons

Status Bar
TAB --> Spaces
Word Wrap
M2 Settings

Also saved are the settings in the Find and Find/Change requesters, ARexx and String Macros, as well as the positions of all requesters.

Advanced features

M2E has several built in features that significantly ease the job of writing Modula-2 code through automation. Correct Case, Complete Word, and Correct Word all use word recognition techniques to reduce the number of errors generated by simple typing mistakes. Bookmarks, the Hex display, and the aforementioned Auto-Indent are features that can ease the programmer's job in a variety of situations.

Correct Case

The Modula-2 language requires that keywords be entered in uppercase characters only. As such, words like "Procedure", "Begin" and "var" are not equivalent to "PROCEDURE", "BEGIN" and "VAR". Case Correct is designed to eliminate the accidental typing mistakes that are often responsible for these simple errors.

When "Config/Case Correct" is selected, M2E monitors user input and automatically converts keywords to all uppercase characters. Thus "Cardinal", "end", and all other keyword case combinations will be changed to their uppercase equivalent after the word is entered.

Example

When Case Correct is selected, the keyword entered as "procedure" is automatically changed to "PROCEDURE" after the word has been entered.

NOTE: Case correction works only for text entry via the keyboard. No case correction of any type is performed on files being loaded from disk.

Complete Word

Complete Word is designed to increase the programmer's coding speed while reducing the potential for errors.

Selecting "Line/Word/Complete Word" automatically finishes any incomplete keyword under or to the left of the cursor.

To complete a word:

1. Place the cursor on, or to the left of, the word to be completed.
2. Select "Line/Word/Complete Word".

M2E analyzes the letters of the word and finds the appropriate match from a supplied list of words. The user must type enough letters to distinguish the intended word from any one with similar spelling that may also be in the list.

Example

In order to use the Complete Word function on "CARDINAL", a minimum of two characters must be entered, counting from the start of the word.

Typing "Ca" and then selecting "Line/Word/Complete Word" or pressing the F7 key, yields the completed word "CARDINAL".

Entire templates have been provided for keywords such as "PROCEDURE", "FOR" as well as others. Completing the letters "PR" does not just yield "PROCEDURE" as one might expect, but rather the entire function and its components. Typing "PR" followed by the F7 key, yields:

```
PROCEDURE ( ) ;  
VAR  
  
BEGIN  
  
END ;
```

If no matches can be found using Complete Word, the "Can't Complete Word, not found!" message is displayed in the status bar. If more than one word can be completed using the number of letters supplied, M2E displays "Can't complete word, not enough characters entered!". Normally, entering one or two more letters will allow M2E to make the proper distinction.

Correct Word

Even the most experienced programmers can fall victim to making ele-

mentary spelling or syntax mistakes in their programs. Correct Word can often be used in conjunction with Complete Word to correct words that Complete Word cannot find due to spelling errors.

To correct a word:

- 1. Place the cursor on, or to the left of the offending text.**
- 2. Select the "Line/Word/Correct Word".**

M2E uses a best-fit algorithm to select the correct spelling and automatically makes the replacement.

Example

Using Correct Word on "form" will modify it to the keyword "FROM". Similarly "PROECDURE" will become "PROCEDURE".

If Correct Word is unsuccessful in its search for a replacement, a "Not found!" message is displayed in the status bar.

The Correct Word function will not work if the first letter of the word is wrong, or the word is too badly misspelled.

Bookmarks

M2E's user definable bookmarks facilitate the process of jumping to and returning from different sections of a file. Bookmarks are convenient temporary place-holders that can ease the task of moving about in a large file.

Up to five bookmarks may be set and recalled at any time according to the following definition:

CTRL + F1..F5	Sets a bookmark at the current cursor position
CTRL + F6..F10	Returns to the bookmark set by the corresponding (F1..F5) function key

To set a bookmark:

- 1. Place the cursor at the desired position.**
- 2. Press the CTRL key along with a function key (F1..F5).**

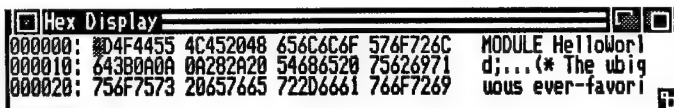
CTRL + F1 Sets bookmark #1 at the current cursor position.

Returning to the bookmark set by CTRL + F1 can be done at any time by pressing CTRL + F6.

NOTE: Bookmarks for a file are valid for the current editing session only and are not saved with the file.

The Hex Display

Selecting "Extras/Show Hex Display" opens a window that shows the hexadecimal equivalent of the area of the file around the cursor. This programmer's tool can be used to view binary files, or as an aid in converting ASCII characters to their hexadecimal equivalents.



In the Hex window, the cursors for the main edit window and the hex display are synchronized together so that any movement in one of the windows is automatically reflected in the other.

The cursor can be repositioned from within the hex display with the cursor keys (up/down/left/right, ALT can be used in conjunction) or the mouse (point and click).

All text entry occurs in overstrike mode and must be entered in hexadecimal.

The window can be resized vertically to see more lines of the file in hexadecimal.

Command summary

The Command summary provides comprehensive and concise explanations of the major functions and commands of M2E. As its title implies, this section of the manual is intended for quick reference use and therefore assumes that you are familiar with M2E's basic editing functions and user-interface conventions.

Keyboard commands

Use the following keys and key combinations to control cursor movement.

NOTE: The "+" and "/" characters used in key sequences are not to be entered directly by the user. They are roughly equivalent to logical operators in the following manner:

"+" = AND

"/" = OR

For example, "SHIFT + Left/Right" means that the user must press and hold a SHIFT key and the Left or Right arrow key.

Cursor movement

Use the following commands to control M2E's cursor movement.

Left/Right	Move by one character
Up/Down	Move by one line
SHIFT + Left/Right	Move by one word
SHIFT + Up/Down	Move by one window
SHIFT + Del	Delete current/next word
ALT + Left/Right	Start/end of line
ALT + Up/Down	Start/end of file
ALT + Del	Delete to end of line
ALT + Backspace	Delete to start of line
CTRL + Left/Right	Move by 10 characters
CTRL + Up/Down	Move by 10 lines
CTRL + Del	Delete current line
CTRL + Backspace	Delete previous line

ENTER Executes a carriage return without breaking the current line

CTRL + SHIFT + Left/Right Moves an entire block of selected text left or right

HOT keys

Use the following key combinations to perform a variety of special M2E operations.

CTRL + ALT + P	Brings the previous M2E window to the front and activates it
CTRL + ALT + N	Brings the next M2E window to the front and activates it
CTRL + ALT + W	Opens a new M2E window
CTRL + ALT + Y	Opens a new M2E window with the file requester up
CTRL + ALT + Q	Forces M2E to unload itself from memory as soon as all the windows are closed

Macros

SHIFT + F1..F10	Insert a String macro
ALT + F1..F10	Run an ARexx macro

Bookmarks

CTRL + F1..F5	Sets a bookmark at the current cursor position
CTRL + F6..F10	Returns to the bookmark set by the corresponding (F1..F5) function key

Menu shortcuts

Use the following key combinations to access menu commands from the keyboard.

⌘ + A	Project/Save As...
⌘ + B	Edit/Mark/Unmark a block
⌘ + C	Edit/Copy
⌘ + D	Line/Word/DownCase Word

A + E Edit/Erase
A + F Search/Find...
A + G Line/Word/Goto Last Modif.
A + H Macros/Stop Recording
A + I Config/Auto-Indent
A + J Line/Word/Jump to Line...
A + K Config/Overstrike
A + L Line/Word/Undelete Line
A + M Macros/Play Macro
A + N Search/Find Next
A + O Project/Open...
A + P Extras/Format Paragraph
A + Q Project/Close Window
A + R Macros/Record Macro
A + S Project/Save
A + T Config/Correct Case
A + U Line/Word/UpCase Word
A + V Edit/Paste
A + W Project/New Window
A + X Edit/Cut
A + Y Project/New & Open
A + Z Edit/Undo Line
A + & Project/Save & Close
A + ? Project/About...
A + = Edit/UpCase
A + - Edit/DownCase
A + / Search/Find/Change...
A + % Macros/Play 'n' Times
A + ! Extras/Center Line/Block
A + @ Extras/Justify Line/Block
A + # Extras/Count Words to EOF
A + \$ Extras/Show Hex Display
A + [Extras/Match Bracket
A + 1 Config/Word Wrap
A + 2 Config/M2 Settings...
A + 3 Config/Save Config

F1	Modula-2/Compile
F2	Modula-2/Link
F3	Modula-2/Run Program
F4	Modula-2/Next Error
F5	Modula-2/Current Error
F6	Line/Word/Correct Word
F7	Line/Word/Complete Word
F8	Line/Word/Toggle Char Case
F9	Line/Word/Swap Chars
F10	Reduce Window

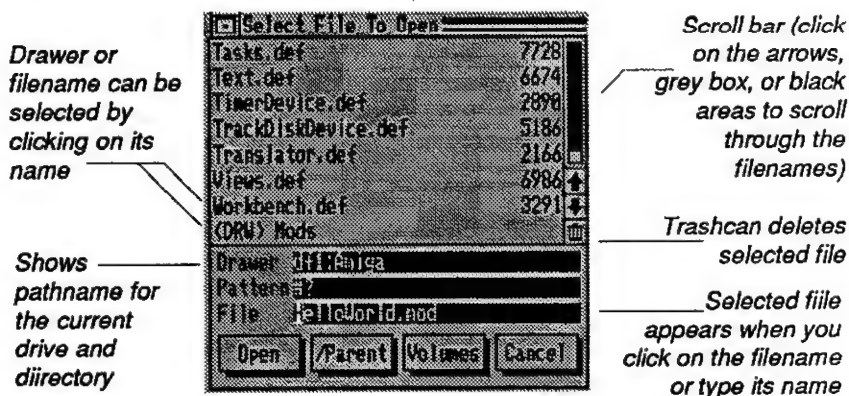
Mouse commands

NOTE: LMB is an acronym for left mouse button.
RMB is an acronym for right mouse button.

Mouse Sequence	Command Description
click LMB	Places the cursor under the mouse pointer.
hold LMB + click RMB	Mark/Unmark a block at the current cursor position
click RMB	Cancel a text selection

The file requester

Whenever a file is opened or saved for the first time, M2E opens a requester so you can choose or specify a filename. The file requester displays an alphanumerical list of files from a specified drawer and allows users to specify a filename for a specific I/O operation. The file requester is used throughout M2E for all file loading and saving operations.



/Parent

Modifies the specified drawer by moving from a subdirectory to its parent. If this function is used when the current drawer is not a subdirectory, the online volumes are listed.

Cancel

Closes the file requester. No files are loaded.

Volume

Lists all AmigaDOS volumes and ASSIGNED directories. Mouse shortcut: click the right mouse button once.

Drawer

The current drawer from which the files are being displayed.

Pattern

Wildcard pattern used to filter files for displaying.

File

Displays the name of the selected file.

A file can be selected by clicking on its name from the list of files, or by typing its name in the "File" string gadget. The selected file can be loaded or saved (depending on the operation) by clicking on the filename and then the "Open" or "Save" gadget, or double-clicking the filename with the left mouse button.

A listing of current on-line volumes and ASSIGNED directories can be viewed by clicking the right mouse button, clicking "Volumes" or by clicking "/Parent" when in a root directory.

The file pattern may be adjusted to display only certain filenames by filtering out the rest. Any ARP or AmigaDOS compatible pattern can be used. Some examples are:

#?	All files
#.MOD	Filenames ending in .MOD

To change the current directory:

- type a pathname in the "Drawer" string gadget
- click on a directory name from the list displayed from the current directory
- click on the "Volumes" gadget show all available volumes and ASSIGNED directories and then click on one of the paths listed

To select a filename:

- type a filename in the "File" string gadget
- click on a filename from the list displayed from the current directory

To load/save a selected filename:

- press the RETURN key after editing the text in the "File" gadget

NOTE: Using the SHIFT key in conjunction with RETURN moves the cursor to the next string gadget (Drawer -> Pattern -> File).

- click the "Open" or "Save" gadget
- double-click on the filename from the list displayed from the current directory

M2E Menu summary

This section documents each function in M2E's menus.

Project

There are ten items in the Project menu.

New Window

Creates a new and empty window for editing.

New & Open

Creates a new and empty window and opens the file requester for loading.

Open...

Reads a file into the active window. User confirmation is required if changes have been made to the file since it was last saved.

Clear

Clears the file from the active window. User confirmation is required if changes have been made since the file was last saved.

Save

Writes the file in the active window to disk under the filename shown in the title bar. If the file is untitled, the operation is identical to the "Project/Save As..." command.

Save As...

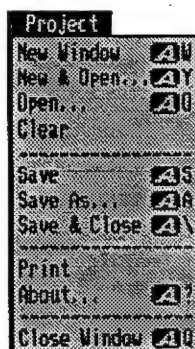
Writes the file in the active window under a new filename. The user specified name replaces the default filename for all future saves.

Save & Close

Writes the file in the active window to disk and then closes its window.

Print

Prints the file in the active window to the system printer. Printing may be interrupted at any time by closing the special "Printing..." window.



NOTE: Printing may continue if the printer or computer system is

equipped with a print buffer or software spooler. Embedded control characters or escape codes in the file may affect the printer's operation.

About...

Displays M2E's version number, copyright message, and the author's name.

Close Window

Closes the active window. User confirmation is required if changes have been made since the file was last saved. This function has the equivalent effect of clicking the close gadget.

Edit

There are eleven items in the Edit menu.

Undo Line

Undoes changes made to the current line. The line is restored to the condition it was in immediately before it was selected as the current line.

Mark

Marks/Unmarks the current cursor position as the starting point of a selected block of text. The size of the selected block can be adjusted with the cursor keys or mouse. Mark can also be done with the mouse by holding the left mouse button and clicking the right mouse button.

Cut

Removes a selected block of text and copies it onto the clipboard.

Copy

Copies a selected block of text onto the clipboard, leaving the original intact in the file and erasing the previous contents of the clipboard.

Paste

Copies the contents of the clipboard into the file at the current cursor position.

Erase

Erases the selected block of text but does not store it on the clipboard.

The "Edit/Erase" command should be used in place of "Edit/Cut" when there is text in the clipboard that must be preserved, but there is something else in the file that must be removed.

NOTE: There is NO way to recover erased text.

UpCase

Converts a selected block of text to uppercase characters only.

DownCase

Converts a selected block of text to lowercase characters only.

Open Clip...

Allows a file to be reads from disk into the clipboard. This information can then be inserted into the current file at any position using the Paste function.

Save Clip As...

Saves the information in the clipboard to a user specified file.

Print Clip

Prints the information in the clipboard to the system printer. For more information on printing see the "Project/Print" command.

Line/Word

There are ten items in the Line/Word menu.

Jump To Line...



Jumps to a user supplied line number. If the specified number is greater than the length of the file, the cursor is placed on the last line.

Undelete Line

Restores a line deleted by a press of the "CTRL + Backspace/DEL" key combination.

UpCase Word

Converts the word under the cursor to uppercase characters.

DownCase Word

Converts the word under the cursor to lowercase characters.

Correct Word

Corrects the spelling of the word under the cursor according to the supplied word list.

Complete Word

Completes the word under the cursor using a best fit algorithm from the supplied word list.

Toggle Char Case

Changes the case of the character under the cursor from upper to lower or lower to upper as the situation dictates and advances the cursor one character.

Swap Chars

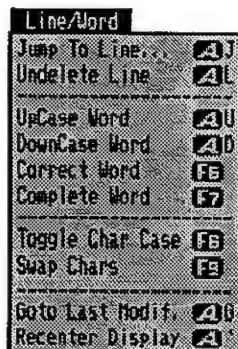
Switches the character under the cursor with its neighbor to the left.

Goto Last Modif.

Moves the cursor to where modifications were last made to the file.

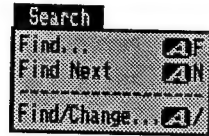
Recenter Display

Puts the current line in the middle of the window.



Search

There are three items in the Search menu.



Find...

The search starts at the current cursor position, can be case (in)sensitive, in the forward or reverse direction, and may be a whole word or just part of a longer one.

Type the string to
be searched for
(RETURN starts)

Toggle case
sensitivity



Toggles word
sensitivity

Modifies search
direction

A successful search places the cursor on the first character of the matched text. A failed search returns "Not Found" on the status bar. Pressing the ESC key or typing CTRL + C during a search aborts the operation.

Find

Search string.

UPPER = lower

- ON Search is case insensitive.
- OFF Search is case sensitive.

Backward

- ON Search will be backwards, from the current cursor position to the top of the file.
- OFF Search will be forwards, from the current cursor position to the bottom of the file.

Whole Word

- ON A matched string must be a word in itself, separated by a valid delimiter.
- OFF A matched string may be a word in itself, or any part of a longer one.

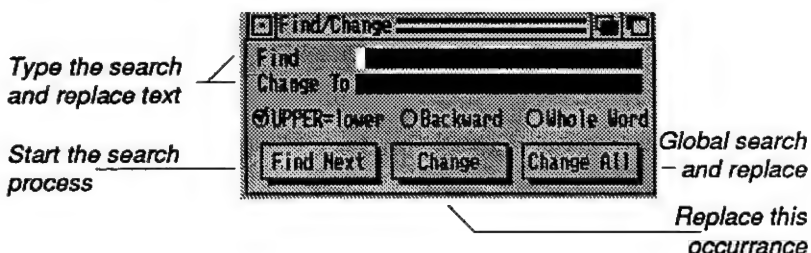
Find Next

Executes a search using the current string and search option settings.

Pressing the ESC key or typing CTRL + C during a search aborts the operation.

Find/Change...

Searches for and replaces with user supplied strings of text. The search starts at the current cursor position, can be case (in)sensitive, in the forward or reverse direction, and must be a whole word or just part of a string. Changes can be made one at a time, or globally.



Find

Search string.

Change To

Replace string.

UPPER = lower

- ON Search is case insensitive.
- OFF Search is case sensitive.

Backward

- ON Search will be backwards, from the current cursor position to the top of the file.
- OFF Search will be forwards, from the current cursor position to the bottom of the file.

Whole Word

- ON A matched string must be a word in itself, separated by a valid delimiter.
- OFF A matched string may be a word in itself, or any part of a longer one.

Find Next

Starts the search process and if successful, places the cursor at

the first character of the matched text.

Change

Replaces the found matched text with the "Change To" string.

Change All

Searches for matching text, replaces it, and then repeats the process until the top or bottom of the file is reached. Use this function with care, incorrect search or replacement strings can ruin a file.

The use of single character wildcards is permitted as long as the number of wildcards in the replacement string not exceed the number in the search string.

The following is an example of a Find/Change using wildcards:

Example

Find ?all ?an
Change To ?ame ?en

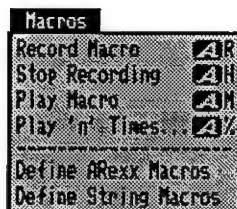
If a Find/Change is carried out using the above Find and Change To strings, the following are possible matches and replacements:

Matched Text	Replaced Text
Tall Pan	Tame Pen
Fall Man	Fame Men
Tall Man	Tame Men
Tall Dan	Tame Den
Mall Man	Mame Men

NOTE: Pressing the ESC key or typing CTRL + C during a search aborts the operation.

Macros

There are six items in the Macros menu.



Record Macro

Records user input (keystrokes, menu commands, mouse clicks etc.) and forms a macro that can be played back using "Macros/Play Macro".

Stop Recording

Halts the macro recording process.

Play Macro

Plays the macro recorded in "Macros/Record Macro".

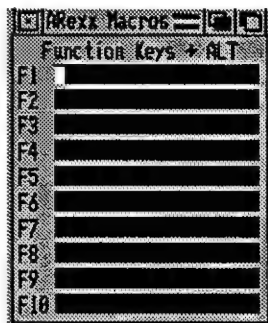
Play 'n' times

Plays the macro recorded in "Macros/Record Macro" any number of times.



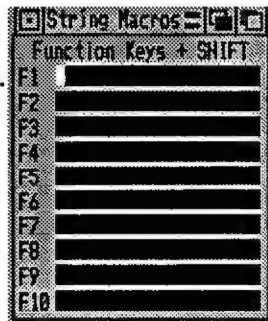
Define ARexx Macros...

Allows up to ten ARexx macros to be defined. These macros can be run using ALT + F1..F10.



Define String Macros...

String macros are created by entering the desired text into any of the string gadgets. They are limited to ASCII characters (control characters are allowed) and can be activated by pressing SHIFT + F1..F10.



Extras

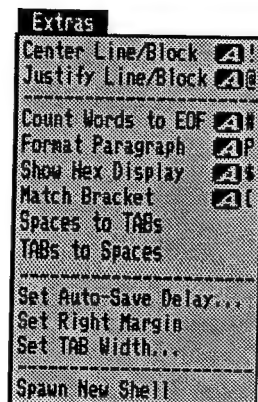
There are twelve items in the Extras menu.

Center Line/Block

Centers a block of text or the current line if no block is highlighted. This operation uses the right margin for its calculations.

Justify Line/Block

Justifies a block of text or the current line if no block is highlighted. This operation uses the right margin setting for its operation.



Count Words to EOF

Tallies the number of words from the cursor forward to the end of the file.

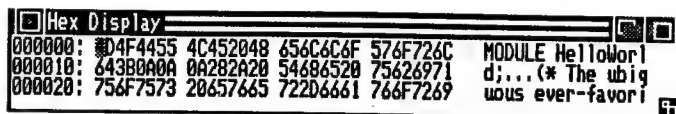
Format Paragraph

Maximizes the number of words per line by splitting and joining subsequent lines as necessary. Carriage returns are inserted in lines longer than the right margin. Words are added, where possible, to lines whose length is less than the right margin.

NOTE: Words are kept intact. A line starting with a space, tab or a carriage return indicates the end of a paragraph.

Show Hex Display

Opens the Hex display window. This display allows the current file to be viewed and edited using hexadecimal. Cursor movement, and scrolling and edits are reflected in both windows in realtime.



Match Bracket

Searches for the mate opening or closing bracket to the one underneath the cursor. Match Bracket is useful for finding errors in equations that use nested brackets.

NOTE: The cursor must be on "{", "[", "(", or ")", "]", ")" to use

Match Bracket.

Spaces to TABs

Searches for and converts lengths of consecutive spaces into equivalent sized TAB characters wherever possible.

TABs to Spaces

Converts TAB characters into the equivalent number of consecutive spaces wherever possible.

Auto-Save Delay...

Allows the user to specify the delay in minutes between automatic saves.



Set Right Margin

Allows the user to adjust the right margin by moving a line to the appropriate column and clicking the left mouse button. Clicking the right mouse button cancels the operation.

Set Tab Width...

Allows the user to specify the width in characters that a TAB represents in the current window.



Spawn New Shell

Creates a new CLI shell.

NOTE: This function requires that the NewShell CLI command be in the C: directory.

Modula-2

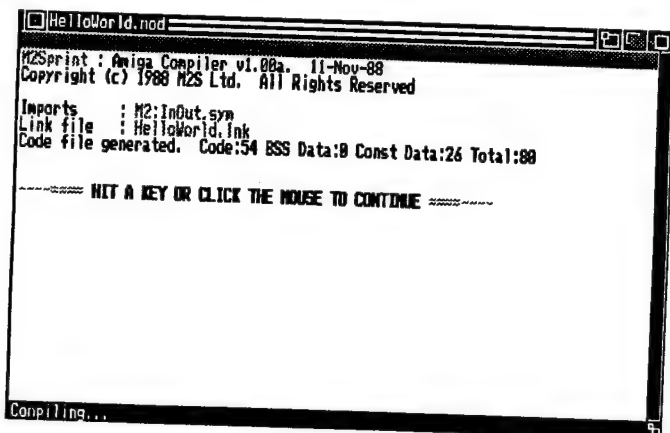
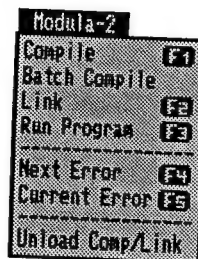
There are seven items in the Modula-2 menu.

Compile

Compiles the file in the current window with the default compiler values in M2Settings.

M2E automatically loads the compiler into memory the first time this function is selected.

The compiler stays resident until the current session with M2E is finished.



Batch Compile

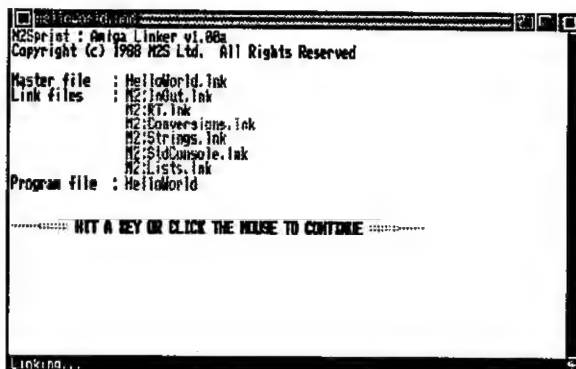
This function automates the process of compiling a group of .DEF and .MOD files. To use Batch Compile, the file in the current window should contain a list of files, with their full pathnames, in the order to be compiled.

After selecting Batch Compile, M2E will orchestrate the process of compiling each file until the list is exhausted. If a compilation error occurs in a definition module compilation stops. If an error occurs in an implementation module, compilation continues.

NOTE: For more information on creating batch files read the section on the M2Batch program in Chapter 14 - "M2Sprint support programs and utilities".

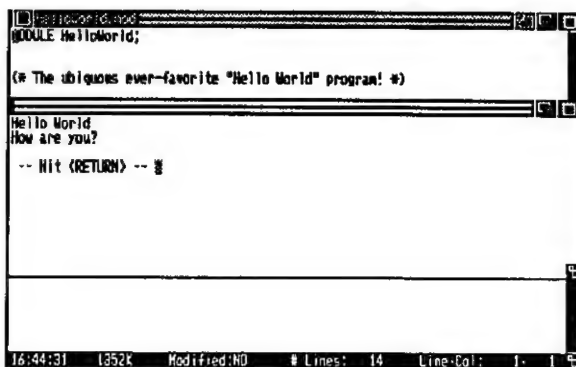
Link

Links the program module as specified in the "Config/M2 Settings" requester.



Run Program

Executes the code generated by the last successful linkage according to the parameters set in M2Settings.



Next Error

Moves the cursor to the next error generated from the last compilation, and displays its error message.

Current Error

Moves the cursor to the current error generated by the last compilation.

Unload Comp/Link

Frees the memory used by the compiler and linker.

Config

There are thirteen items in the Config menu.

ARexx Console

- ON Open a standard console window when running an ARexx macro. Useful when debugging ARexx macros.
- OFF No console window is opened.

Auto Indent

- ON New lines have the same indentation as the line above.
- OFF New lines have no indentation.

Auto-Save

- ON M2E automatically saves the file in the current window. Intervals can be adjusted by selecting "Extras/Auto-Save Delay...". Auto-Saves are only performed on files that have been modified since the prior save.
- OFF Files are not automatically saved.

Case-Correct

- ON Monitors keyboard entry and corrects any keyword that is entered in the wrong case.
- OFF Ignores keyword case mistakes.

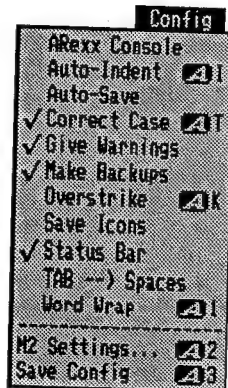
Give Warnings

- ON When closing the file in the current window or opening a new one, the program will ask for confirmation if changes made have not been saved.
- OFF The program will never ask for confirmations.

Make Backups

- ON The old version of the file is preserved under ":T/M2E.backup".
- OFF No backups are made.

NOTE: Backups only occur if a T directory exists on the same volume the save is to occur.



Overstrike

- ON As new text is entered, text under the cursor is erased and replaced.
- OFF As new text is entered, text after the cursor is pushed to the right.

Save Icons

- ON Icons are generated and saved with files.
- OFF No icons are saved.

Status Bar

- ON Status Bar is displayed.
- OFF No Status Bar.

TAB --> Spaces

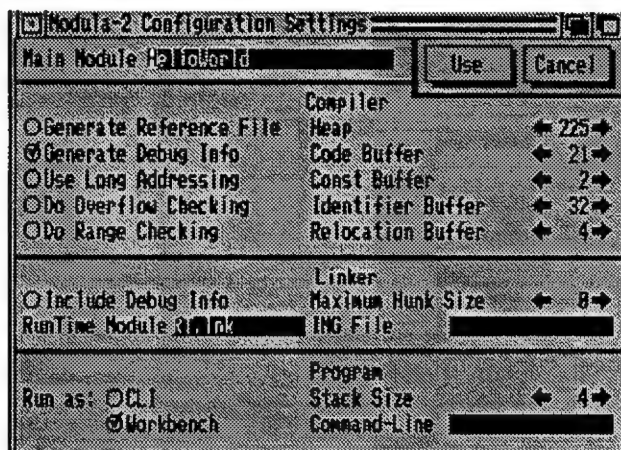
- ON The TAB key inserts the equivalent number of spaces instead of a tab character.
- OFF TABs are preserved.

Word Wrap

- ON Carriage returns are automatically inserted before the current word when the cursor reaches the right margin.
- OFF Typing can continue beyond the right margin.

M2 Settings

Brings up the Modula-2 Configuration Settings requester and allows default compiler and linker settings to be adjusted.



Compiler

Compiler options can also be adjusted by embedding compiler switches inside a program's source code. For more information on how to use compiler switches see Chapter 12 - "M2C - The Compiler".

Generate Reference File

Creates an extra reference file required for use with M2Debug. This option must be on during compilation if any debugging is to be performed with the M2Sprint debugger.

Generate Debug Info

Includes extra information for use during debugging. M2Debug and other symbolic debuggers (such as Metadigm's "Metascope") can only be used on parts of the program that has been compiled with this option on.

All M2Sprint modules have been compiled with this option on for ease of debugging. This is equivalent to the \$D compiler switch.

Use Long Addressing

Use 32 bit addressing instead of the default A4 relative addressing. This is equivalent to the \$L compiler switch.

Do Overflow Checking

Traps overflow errors caused by the program exceeding the upper or lower bounds of a scalar-type variable (CARDINAL, INTEGER, and REAL). This is equivalent to the \$O compiler switch.

The following example generates a run-time error:

```
...  
VAR  
    x, y : CARDINAL;  
BEGIN  
    x := 1000;  
    y := x * x;  
...
```

Do Range Checking

Allows range checking to be enabled or disabled. When enabled, the compiler attaches code to ensure that arrays are used within their bounds. This is equivalent to the \$T compiler switch.

The following example generates a run-time error when "i > 10":

```
...
VAR
    i          : CARDINAL;
    numbers    : ARRAY [0..10] OF CARDINAL;
BEGIN
    FOR i := 0 TO 15 DO
        numbers[i] := i;
    END;
END;
...
```

Heap

General purpose memory used by the compiler.

Code Buffer

Temporary storage area used by the compiler to store code before writing it to a file.

Const Buffer

Used to store strings.

Identifier Buffer

Used to store identifiers.

Relocation Buffer

Used when code is IMPORTed from other modules. The more importation being done, the larger the relocation buffer must be.

Linker

Include Debug Info

Includes any debugging information that may have been included at compile time. Programs that are linked with this option on will take up more disk space.

RunTime Module

The runtime module to be linked with the executable.

Maximum Hunk Size

Determines the number of hunks that the executable will be made up of. All programs are composed of one or more hunks of code.

There is a certain amount of overhead associated with each hunk, therefore programs made up of many hunks tend to be larger and take longer to load.

Although programs with fewer hunks are smaller and load faster, they require large contiguous sections of memory since hunks cannot be subdivided in memory. Dividing the program up into more hunks means that the system can use "scatter loading" techniques to utilize small isolated sections of memory to store the program.

A hunk size value of zero means that the program will be one hunk. A hunk size of one means that there will be one hunk per module. Hunk sizes above one indicate the maximum kbytes/hunk (e.g. A value of 15 means each hunks will be a maximum of 15 kbytes).

IMG File

File containing the list of images to be linked to the executable. (refer to the discussion of the IFF2Obj program in Chapter 14 - "Support programs and utilities").

Program

The following options are for simulation and testing purposes only. None of these options have any effect on the program being generated and do not imply that the final executable can be run from only Workbench or CLI, have a static stack size, or be always passed certain command-line arguments.

Main Module

Specifies the name of the main program module.

Run as CLI/Workbench

Specifies whether the program launching should be a

simulation of CLI or Workbench. This allows programmers to test their startup code for argument handling etc.

Stack Size

Sets the stack size to be used when the program is run with "Modula-2/Run Program". Has the equivalent function as setting the stack size from CLI or in an application's icon.

Command-Line

Arguments to be passed to the program when run with "Modula-2/Run Program". Has the equivalent function as passing arguments to a program from CLI.

Save Config

Saves M2E's current configuration to the files
"M2Data:M2.editconfig", "M2Data:M2.compconfig",
"M2Data:M2.linkconfig".

12. M2C - The Compiler

M2C is a high speed single-pass compiler with a full implementation of the Modula-2 programming language. Its job is to translate Modula-2 source files into executable code, runnable by the Amiga's 68000 processor.

The M2Sprint compiler processes four different types of source files:

- Modula-2 program modules (filenames have the .mod extension).
- Modula-2 implementation modules (filenames have the .mod extension).
- Modula-2 definition modules (filenames have the .def extension).
- Batch files containing a list of filenames to be compiled.

NOTE: All of the source files accepted by the compiler must be in ASCII format as produced by M2E or other Amiga text editors such as Commodore's ED or MicroEmacs.

Depending on the type of source file being compiled, the M2Sprint compiler will generate one of the following types of files:

- Link files (filenames have the .lnk extension) contain MC680x0 machine code and are produced by compiling program or implementation modules.
- Symbol files (filenames have the .sym extension) are produced by compiling a definition module and contain a compact binary representation of the source file.
- Reference files (filenames have the .ref extension) contain information for use with M2Debug, M2Sprint's source-level debugger. Reference files are generated when compiling implementation or program modules, only if the option has been turned on.

Compiler options

The M2Sprint compiler offers several options which control how much memory the compiler requires and how executable code is generated. These options can be controlled in various ways. This section describes the individual options themselves, following sections will describe how to adjust these options from the different environments available.

Memory usage

Compilation is a process which generally requires substantial amounts of memory. The size of the memory buffers used by the compiler can be adjusted to suit various situational requirements, however in most cases they can be left at their default settings. When compiling very large programs or when available memory is low it may become necessary to increase or decrease the buffer sizes.

NOTE: With the exception of the "Disk Buffer", the compiler's performance is not affected by the size of the buffers used.

The stack must be set to at least 16K

The following buffers are used by the compiler:

Heap

This is general-purpose memory used during compilation to store internal tables and lists.

Range : 50K to 999K
Default : 80K

Code Buffer

The compiler deposits generated code in this buffer before writing it to disk.

Range : 5K to 32K
Default : 10K

Disk Buffer

This memory is used to buffer read/write operations to the file system. Generally, the larger the buffer, the faster the compilation.

Range : 1K to 32K
Default : 1K

NOTE: The compiler may require up to three disk buffers at any given moment, so if the Disk Buffer size is set to 3K, the compiler may take a total of 9K of system memory for its disk buffers.

Identifier Buffer

This memory buffer is used to store all user-defined and imported identifiers encountered during compilation.

Range : 5K to 32K
Default : 20K

Relocation Buffer

This memory buffer is used to store relocation information during compilation. A relocation entry is generated whenever a reference is made to a variable or a procedure belonging to a different module (an imported variable or procedure) or when accessing global variables using long addressing.

Range : 1K to 32K
Default : 3K

String Buffer

This memory buffer is used to store string constants during compilation.

Range : 1K to 32K
Default : 3K

Code generation options

These options affect how the compiler produces code and what happens at code generation time.

Reference File Generation

Reference files are used by the M2Sprint debugger, "M2Debug". They contain information to allow the debugger to pin-point the exact location of a program crash within the source code. Reference files are only generated when compiling program or implementation modules, not definition modules. When performing any debugging, it is advisable to generate reference files for every module compiled.

Compiler Beep

The M2Sprint compiler can beep whenever it has finished a compila-

tion. This is very useful when performing long compilations in the background while other work is being done.

Icons

The M2Sprint compiler can generate icons for the files it produces. The compiler uses the file "M2.link.info" from the M2Data: drawer as a template when creating a new icon. That is, an exact duplicate of the "M2.link.info" file is created under the name of the file the compiler is generating.

Input Drawer

This option allows to specify in which drawer the compiler is to look for source files. By default, all source files are obtained from the current drawer.

Output drawer

This option allows to specify in which drawer the compiler is to put all of the files it generates. By default, all files are put in the current drawer.

Short/Long Addressing

Typically, Modula-2 programs access global data by relative offsets from the A4 68000 register. This is often the fastest and smallest method possible to access data. Every module in a program requires a different value for A4. As such, every Modula-2 procedure has to reload the A4 register upon entry, and restore the original value of A4 upon exit.

It is possible to access global data using absolute addressing. This will generally yield larger and slower code than A4 relative addressing. The advantage to absolute addressing is that it does not require A4 to be saved and restored in every procedure.

Long addressing can often be used to good advantage for procedures that never (or seldom) access global module data. Note that references to variables from other modules are always done using long addressing. Only variables from the same compilation unit can be accessed using A4 relative addressing.

Overflow Checking

This option allows the compiler to generate extra code to trap any program errors caused by the program exceeding the maximum or minimum values allowed by a variable.

Overflow checking is performed during program execution, and not

during compilation. For this reason, it is often referred to as "run-time checking".

When using this option, programs will be larger and slightly slower than usual, but this option is very useful in the debugging phase of a program. When an overflow error occurs, it is possible to activate the M2Sprint debugger and investigate the cause of the error. An example of the type of error overflow checking can trap is:

```
VAR
    x,y : CARDINAL;

BEGIN
    x := 1000;
    y := x * x; (* exceeds the max value
                that can be stored in a CARDINAL *)
```

Range Checking

This option allows the compiler to generate extra code to trap any program errors caused by the program exceeding the upper or lower bounds of an array or subrange. For example, trying to access the 100th element of a 99 element array.

Range checking is performed during program execution, and not during compilation. For this reason, it is often referred to as "run-time checking".

When using this option, programs will be larger and slightly slower than usual, but this option is very useful in the debugging phase of a program. When a range error occurs, it is possible to activate the M2Sprint debugger and investigate the cause of the error. The following example generates a range checking run-time error when "i > 10":

```
VAR
    i      : CARDINAL;
    numbers : ARRAY [0..10] OF CARDINAL;

BEGIN
    FOR i := 0 TO 15 DO
        numbers[i] := i;
    END;
END;
```

Adjusting the default settings

The default configuration settings used by the M2Sprint compiler are located in the file M2Data:M2.compconfig. There are two ways to modify the contents of this file:

- by selecting the "Config/Compiler Settings..." command from within the integrated environment.
- by using the CompSettings tool from CLI or Workbench.

Please refer to Chapter 14 - "Support programs and utilities" for more information on how to use these requesters.

Compiler switches

Compiler switches control the various options outlined in the "Compiler Options" section above, along with a few extra features, by directly including compiler directives in the source code.

Compiler switches have the following syntax:

```
Switch = "$" Letter [Letter| "+" | "-" | "="]
```

The compiler expects switches to be located in a comment as in this template:

```
(*Switch {,Switch}*)
```

The following example would turn range checking on and overflow checking off:

```
(*$R+, $O-*)
```

The following switches are recognized by the M2Sprint compiler. All other switches are ignored. Options can be turned on, turned off, or can be reverted to the default.

To turn a switch on, specify the switch letter followed by a "+" character. To turn it off, you follow the switch with a "-" character. To revert the option to the current default, you specify the "=" character. Certain switches do not require any of these characters and are activated simply by including the switch name by itself.

\$I - No Implementation Module

This option is used in definition modules. When the option is specified, it tells the compiler that there is no implementation module accompanying this definition module. This is used mostly for the Amiga interface modules. The compiler will report an error if the definition module includes any variable declarations, string constant declarations, or non-INLINE procedures (see below for more information on INLINE).

\$L - Long Addressing on/off/default

This option controls the use of long and short addressing. When this option is turned on using the \$L+ switch, all following procedures will use long addressing. Specifying \$L- reverts to A4 relative addressing. \$L= reverts to the current default.

\$MC - Module data to Chip memory

\$MF - Module data to Fast memory

These two options control the same feature. When either of these switches is specified, it forces all the global variables of a module to be allocated in a particular type of Amiga memory. The default is to allocate the variables in Public memory. The \$MC switch forces the data to go to Chip memory, and the \$MF switch forces it to go to Fast memory. All the data from a given module must go in the same type of memory. Because of this, only one of these switches is accepted per module.

\$O - Overflow checking on/off/default

This option is useful to control exactly where overflow checking is performed. There are certain instances when overflow checking must be turned off, otherwise some unwanted errors will occur. Using the \$O- option, overflow checking can be disabled for a specific section of code. When the "dangerous" section is past, the option can be reverted to the current default using the \$O= switch. \$O- turns overflow checking off, \$O+ turns it on, and \$O= reverts to the current default for the option.

\$R - Range checking on/off/default

This option is useful to control exactly where range checking is performed. There are certain instances when range checking must be turned off, otherwise some unwanted errors will occur. Using the \$R- option, range checking can be disabled for a specific section of code. When the "dangerous" section is past, the option can be reverted to the current default using the \$R= switch. \$R- turns range checking off, \$R+ turns it on, and \$R= reverts to the current default for the option.

\$\$ - Don't copy string parameters on the stack

This option disables the copying on the stack of value string parameters. This produces much faster and smaller code. This option does not affect VAR parameters, only value parameters. \$\$ should be used with care as it may cause valid Modula-2 programs not to function.

The basic rule is that when a string is passed by value, it can in theory get modified within the procedure without affecting the parameter's value. If you know that a procedure does not modify a string, it only reads from it, then specifying this option will cause much faster execution. For example, the standard Strings module uses this option extensively. Specifying the \$\$ switch will affect the next procedure declared only, subsequent procedures will function normally.

\$X - Procedure entry/exit code generation off

When generating the code for a procedure, the compiler generally puts a small chunk of code at the start and at the end of the procedure to handle parameter passing and stack usage for the procedure. It is sometimes desirable to disable the generation of this code. Specifying the \$X switch will cause the next procedure declared to have no entry/exit code generated. This switch is active for one procedure only, subsequent procedures will function normally.

Evaluation of compiler options

The M2Sprint compiler can take its options from the various locations described in previous sections. This short section simply exposes in which order the compiler interprets the options. The last option evaluated always takes precedence over any previous setting.

- The compiler sets its configuration to a fixed default.
- The compiler tries to read the M2Data:M2.componfig file and adjusts its configuration accordingly if the file can be loaded. If the compiler is started from the editor, then the compiler uses the current settings provided by the editor.
- The compiler interprets any compiler options provided on the command-line when started from CLI or Workbench.
- The compiler interprets any outstanding ARexx commands.
- The compiler interprets any options given on every line of a batch

compilation file.

- The compiler finally interprets any switches specified in the source code.

This means that the switches specified directly in the source code override all other settings.

Version control system

With most implementations of older languages such as "C", it is possible to link together old and new versions of object files. This can result in a program which does not work, and a lot of time spent trying to find the cause of the problem. M2Sprint supports Modula-2's standard version control system that prevents incompatible modules from being linked together.

Version control is automatically handled by the compiler and linker through identification (ID) codes. Each time The M2Sprint compiler translates a definition module into a symbol file, it assigns a randomly generated ID code to it. When the implementation counterpart of that module is compiled, the compiler automatically reads the symbol file and assigns the same ID code to the link file. No matter how many times the implementation module is compiled, it will always have the same ID code as its corresponding symbol file.

If the symbol file is recompiled, it is assigned a new ID code. At this point, the ID code in the symbol file and in the link-files are different, since the link file still has the same ID code as the previous version of the symbol file. This is when a version conflict arises.

Another cause of version conflicts is when several modules import another given module, and the definition part of that module is recompiled, without recompiling all the modules that import from it. For example, if module A and module B both import something from module C and the definition module for module C is recompiled, then both module A and module B should also be recompiled because they depend on module C. If this is not done, a version conflict error occurs.

The linker may detect version conflicts where the compiler does not. Files that import elements from an older version of a module will also cause a version conflict. The compiler cannot detect these types of errors because it does not import symbol modules that may have used elements from an older version of the symbol module.

To fix a version conflict problem, it is best to simply recompile all the modules composing the program from scratch.

Using the compiler

Modula-2 source files can be compiled with M2C at any of the following locations:

- from the M2E environment
- from CLI
- from Workbench
- from an ARexx macro

Compiling from the environment:

1. **Open a new M2E editing window and load the source file to be compiled.**
2. **Select the "Modula-2/Compile" command.**

If the current compilation is the first, M2E must take the time to load the compiler from disk. The compiler is searched for as "M2C". If it is not found under that name, an attempt is made to load it as "M2Sprint:M2C".

Once loaded, the compiler will remain in memory and will not need to be reloaded for future compilations. The memory used by the compiler may be freed using the "Modula-2/Unload Comp/Link" command.

Compiling from CLI:

1. **Type "M2Sprint:M2C FileName" at a CLI prompt where FileName is the name of the Modula-2 file to be compiled.**
2. **Press RETURN.**

As all other M2Sprint tools, the M2Sprint compiler can be run from CLI as a normal Amiga application. The following command-line template is supported by the compiler:

```
FILES/... , TO/K, HEAP/K, CODE/K, DISK/K, ID/K, RELOC/K,  
  STRING/K, RF=REF_FILE/S, LA=LONG_ADDR/S,  
  CO=CHK_OVERFLOW/S, CR=CHK_RANGE/S, FREQ/S,  
  BEEP/S, HOT/S
```

This template can be obtained from CLI by starting the compiler with a question mark ("?.") as only parameter. The template for the compiler works exactly like normal AmigaDOS templates.

Any parameters passed to M2C from CLI will be used in place of the compiler's settings as defined in M2Data:M2.compconfig. If a switch such as "BEEP" is included in the command-line, it will toggle the value specified in M2.compconfig. If BEEP is turned on in M2.compconfig, including it in the command-line will turn the beeping off, and vice-versa. Here is a description of the various command-line options available:

FILES/...

Allows you to specify any number of files to compile. Multiple filenames may be specified by separating each with a space. Filenames containing spaces must be surrounded by quotation marks. Filenames can include complete directory paths. When compiling an implementation module, it is not necessary to specify the .mod extension, the compiler will append it automatically.

TO/K

Allows you to specify a destination directory where the compiler is to output all of its files. If this is not specified, the compiler will output its files to the "Output Drawer" specified in the M2.compconfig file. If the "Output Drawer" is empty, then output will go to the current directory.

HEAP/K

Allows you to override the current default for the size of the compiler heap. Specify this parameter followed by a number in the range [50..999].

CODE/K

Allows you to override the current default for the size of the compiler code buffer. Specify this parameter followed by a number in the range [5..32].

DISK/K

Allows you to override the current default for the size of the compiler disk buffer. Specify this parameter followed by a number in the range [1..32].

ID/K

Allows you to override the current default for the size of the com-

piller identifier buffer. Specify this parameter followed by a number in the range [5..32].

RELOC/K

Allows you to override the current default for the size of the compiler relocation buffer. Specify this parameter followed by a number in the range [1..32].

STRING/K

Allows you to override the current default for the size of the compiler string buffer. Specify this parameter followed by a number in the range [1..32].

RF=REF_FILE/S

Allows you to invert the current setting of the "Generate Reference File" option.

LA=LONG_ADDR/S

Allows you to invert the current setting of the "Use Long Addressing" option.

CO=CHK_OVERFLOW/S

Allows you to invert the current setting of the "Do Overflow Check" option.

CR=CHK_RANGE/S

Allows you to invert the current setting of the "Do Range Check" option.

FREQ/S

Allows you to invert the current setting of the "Use File Req" option.

BEEP/S

Allows you to invert the current setting of the "Beep After Compiling" option.

HOT/S

Allows you to start the compiler as an ARexx server. Once started as a server, the compiler can be called directly using ARexx commands. It can be aborted and unloaded from memory by hitting CTRL-C in the same CLI window where it was started from, or by using the AmigaDOS BREAK command. It can also be aborted by sending it the ARexx "QUIT" command. Refer to the section on compiling from ARexx below for more information on the compiler's ARexx interface.

Examples:

```
M2Sprint:M2C mySource heap 222 beep
```

This will cause the compiler to compile the file "mySource.mod", allocating a compiler heap space of 222K, and toggling the setting for the Beep option.

```
M2Sprint:M2C mySource1 mySource2 id 32 hot
```

This will cause the compiler to compile the files "mySource1.mod" and then "mySource2.mod" using an identifier buffer of 32K. Once the compilation finished, the compiler will then enter ARexx server mode due to the presence of the HOT option.

Compiling from Workbench:

Individual or multiple files can be selected for compiling by shift-clicking their icons. Shift-double-clicking M2C's application icon will launch M2C and compile each source file selected in the order they were selected.

1. Find the file(s) to be compiled.
2. Select the file(s) to be compiled by clicking their icons while holding down the SHIFT key.
3. After the file(s) have been selected, double-click on the M2C icon while holding down the SHIFT key.

If the M2C icon is selected by itself, without shift-clicking any other files, the compiler will open a window and present the command-line template described in the section "Compiling from CLI" above. The compiler will then behave exactly like if it were started from CLI. You can enter multiple filenames and enter switches.

Compiling from an ARexx macro:

The M2Sprint compiler can be used as an ARexx server. This allows it to be used with any ARexx-compatible text editor. The compiler must be started with the HOT option explicitly specified (see "Compiling from CLI" above) in order to use it as a server.

Once started in server mode, the compiler opens a message port under the name "M2C_ARexx_Port" which can be used to send ARexx commands to it. Sending any of the configuration commands to the compiler changes its internal defaults and will affect all following compilations. A list of the supported ARexx commands and their respective parameters follows:

COMPILE filename

Sending this command to the compiler will cause it to compile the file <filename> or <filename.mod>. The filename can include a directory path. Note that the compiler's current directory is the one from which it was started, and not the directory from which the ARexx command was invoked.

TO dirname

Allows you to specify a destination directory where the compiler is to output all of its files. If this is not specified, the compiler will output its files to the "Output Drawer" specified in the M2.comconfig file. If the "Output Drawer" is empty, then output will go to the current directory.

HEAP size

Allows you to override the current default for the size of the compiler heap. Specify this parameter followed by a number in the range [50..999].

CODE size

Allows you to override the current default for the size of the compiler code buffer. Specify this parameter followed by a number in the range [5..32].

DISK size

Allows you to override the current default for the size of the compiler disk buffer. Specify this parameter followed by a number in the range [1..32].

ID size

Allows you to override the current default for the size of the compiler identifier buffer. Specify this parameter followed by a number in the range [5..32].

RELOC size

Allows you to override the current default for the size of the compiler relocation buffer. Specify this parameter followed by a number in the range [1..32].

STRING size

Allows you to override the current default for the size of the compiler string buffer. Specify this parameter followed by a number in the range [1..32].

REF_FILE [ON|OFF]

Allows you to adjust the current setting of the "Generate Reference File" option.

LONG_ADDR [ON|OFF]

Allows you to adjust the current setting of the "Use Long Addressing" option.

CHK_OVERFLOW [ON|OFF]

Allows you to adjust the current setting of the "Do Overflow Check" option.

CHK_RANGE [ON|OFF]

Allows you to adjust the current setting of the "Do Range Check" option.

FREQ [ON|OFF]

Allows you to adjust the current setting of the "Use File Req" option.

BEEP [ON|OFF]

Allows you to adjust the current setting of the "Beep After Compiling" option.

Batch compilation

Large Modula-2 programs can be made up of dozens of individual source files that make up the different modules the main program is dependent on. Due to the number of files involved, compiling large applications from start to finish can take a considerable amount of time if done by invoking M2C for each individual file to be compiled.

The M2Sprint compiler provides a feature known as Batch Compilation to help ease this problem. The compiler's Batch Compilation feature allows groups of Modula-2 source files to be compiled quickly and easily.

A compiler batch file consists of a list of filenames with optional parameters after each filename. As the compiler processes the batch file it compiles each file listed.

Although the files specified in the batch file can be any Modula-2 source file, they must be given in the order of their priority. So that the batch compilation is successful, the modules must be listed (and therefore compiled) in accordance to Modula-2's rules of dependence. Modules that do not depend on other units in the batch file should be put before those that do. In other words, all the modules that a given file depends upon (IMPORTs) must be listed in the batch before that file.

Every line of a batch file should contain the name of a single source file. The line can also contain any compiler options in the same format as if the compiler was being launched from CLI.

NOTE: Comments can be included in the batch file by putting a semi-colon ";" at the start of a line, followed by the comment text.

Creating batch files

Batch files can be created with M2E or any other text editor by listing the names of the files to be compiled in proper order. Care must be taken so that definition modules (.DEF) are listed before their implementation module (.MOD) counterparts.

The M2Batch tool automates the process of creating batch files. After analyzing the IMPORT statements of a Modula-2 source file, M2Batch generates a file containing the list of files the program is dependent on, in the order they need to be compiled.

M2Batch accepts as input a Modula-2 program source file and will create a batch file using the same filename with the ".bat" extension.

Examples of batch files created by M2Batch can be found throughout the Demos disk. Refer to Chapter 14 - "Support programs and utilities" for more information on using M2Batch.

To use Batch Compilation from the environment, load the batch file in the editor, and select the "Modula-2/Batch Compile" menu command. This will start the compiler going.

To use Batch Compilation from CLI, you must use IO redirection. For example, if the batch file is called batch.bat, you must start the compiler from CLI using a command-line such as:

```
M2C <batch.bat
```

To use Batch Compilation from Workbench, you must start the compiler by itself by double-clicking its icon. You then use IO redirection as if it was started from CLI.

Batch compilation cannot be invoked through the ARexx interface.

Locating symbol files

When compiling a module, the compiler will often require access to symbol files. Everytime a module imports (IMPORT) another module, the imported module's symbol file must be accessed.

Typically, the compiler will look for symbol files in the current drawer, if not found, it will then look in the M2: drawer. If still not found, it will present a file requester prompting the user to select the file.

The M2Sprint compiler supports a special feature that allows it to scan several different drawers for its symbol files. The file M2.searchpath found in the M2Data: drawer contains a list of all the drawers that should be searched. This is very similar in concept to the AmigaDOS PATH command.

As an example, if you have most of your symbol files in a RAM disk to speed up compiling, but have not enough room to store all of them there. You could copy only the more used files to the ram disk and leave the less commonly referenced files on disk. You would then use M2.searchpath to tell the compiler that certain files are on disk. M2.searchpath might look like:

```
RAM:MainM2
DH0:ExtraM2
```

with a M2.searchpath file defined as above, the compiler would first try to find symbol files by looking in the current drawer. If it's not there, then it would look in the M2: drawer. If still not found, then it would look in RAD:MainM2, and finally in DH0:ExtraM2.

Every line of the file indicates a specific search path. The path specification may be relative to the current drawer.

Compiler Output

After a successful compilation of a program module or of an implementation module, the compiler reports certain information regarding the file it has just compiled. Here is a list of the information which is displayed along with an explanation of every value:

Code

This value indicates exactly how many bytes of machine code have been generated during the compilation.

Global Data

This value indicates exactly how many bytes have been allocated for the current module to store global variables.

String Data

This value indicates exactly how many bytes are used by the current module to store strings.

Reloc Data

This value indicates how many bytes are used in the current module to store relocation information. Relocation information will always become part of the final executable, thus requiring disk space, but it is discarded at the time the executable is loaded in memory when it is executed.

Total

This value is simply the sum total of the previous four values. Note that this value does not correspond to the actual file size of the file output by the compiler.

Compilation errors

After compiling a module, the compiler may report it has found errors during compilation. These errors are often due to typing mistakes, or incorrect language syntax. The compiler specifies how many errors it encountered during compilation and generates a file containing a list of all the errors encountered along with the position in the source file where the errors were detected. The file generated is called "T:M2.errorlog" and is in binary format.

The contents of the error log file can be interpreted in two ways. Using the integrated environment, you simply need to type F4 to step through all the compilation errors. From CLI or Workbench, the M2Errors utility can be used to display all the errors. For more information on finding errors with M2Errors or M2E refer to:

Chapter 8 - "Using the environment"

Chapter 11 - "M2E - The editor"

Chapter 14 - "Support programs and utilities"

A complete listing of compiler errors and descriptions is given in Appendix A - "Error messages and return codes".

The pseudo module SYSTEM

Modula-2 defines the SYSTEM module as being a pseudo-module supported by the compiler containing implementation specific details. M2Sprint's SYSTEM module adheres to this definition, and provides the services expected from SYSTEM, as described by the Modula-2 language definition.

SYSTEM allows programmers to access processor dependant features, allowing for efficient low-level system programming. Having a SYSTEM module lessens the incompatibilities caused by different processing units on different machines. Note that the contents of the SYSTEM module are inherently implementation-dependant, and thus the use of the module should be kept to a minimum if portability is an issue.

The SYSTEM module is unlike any other M2Sprint module. Although elements can be imported from SYSTEM it has no formal definition or implementation part because SYSTEM is actually "resident" in the compiler itself.

The following is an attempt at defining SYSTEM using standard Modula-2 syntax. This is not completely accurate due to the module's unique nature, but it does give a good idea of what services are provided by SYSTEM.

MODULE SYSTEM;

TYPE

```
BYTE; (* A 680x0 byte, 8 bits *)
WORD; (* A 680x0 word, 16 bits *)
LONGWORD; (* A 680x0 longword, 32 bits *)
ADDRESS = POINTER TO BYTE;
```

```
STREPTR = POINTER TO ARRAY [0..30000] OF CHAR;
```

```
PROCEDURE ADR(VAR object:AnyType): ADDRESS;
PROCEDURE TSIZE(AnyType): INTEGER;
PROCEDURE SHORT(long:LongType): ShortType;
PROCEDURE LONG(short:ShortType): LongType;
PROCEDURE LONG(hi,lo:ShortType): LongType;
PROCEDURE SHIFT(val:AnyType; int:INTEGER): AnyType;
```

```
PROCEDURE CODE(value:CARDINAL,...);
PROCEDURE REGISTER(register:CARDINAL): ADDRESS;
PROCEDURE SETREG(register:CARDINAL; val:AnyType);
PROCEDURE LOADREGS(registers:BITSET);
PROCEDURE SAVEREGS(registers:BITSET);
PROCEDURE GETA4();
```

```

PROCEDURE TERMPROC (procedure:PROC);

END SYSTEM.

```

Here is a description of the various components of the SYSTEM module:

BYTE

BYTE is a special type exactly the size of a 68000 byte (8 bits), thus it can hold values ranging from 0 to 255. This is the smallest addressable memory unit on a 68000 processor. No operations can be performed on values of type BYTE, except assignment. Any value which is held in memory within exactly 8 bits can be directly assigned to BYTE. This includes CHAR and small SETs.

BYTE will typically be used to create a parameter of type ARRAY OF BYTE in procedures. This allows the procedure to accept any type of parameter and map the parameter into an array of bytes. This replaces ARRAY OF WORD used on other implementations of Modula-2, since the BYTE is the smallest addressable unit on a 680x0 processor.

For example, BYTE can be used such as:

```

...

VAR
  byte : BYTE;
  char : CHAR;
  set  : SET [0..7];
  size : CARDINAL;

BEGIN
  byte:=char;
  byte:=set;
  char:=byte; (* this will not compile *)
  set :=byte;  (* this will not compile *)
  size:=SIZE(BYTE); (* this equals 1 *)
...

```

WORD

WORD is a special type exactly the size of a 68000 word (16 bits), thus it can hold values ranging from 0 to 65535. No operations can be performed on values of type WORD, except assignment. Any value which is held in memory within exactly 16 bits can be directly assigned to WORD. This includes CARDINAL, INTEGER, BITSET and SETs. Variables of type WORD must be word aligned in

memory. This is a limitation of the 68000 processor. The compiler will normally ensure that this alignment is done.

WORD will typically be used to create a parameter of type ARRAY OF WORD in procedures. This allows the procedure to accept any type of parameter at least 16 bits in size and map the parameter into an array of words.

For example, WORD can be used such as:

```
...
VAR
    word : WORD;
    card : CARDINAL;
    int  : INTEGER;
    set  : BITSET;
    size : CARDINAL;

BEGIN
    word:=card;
    word:=int;
    word:=set;
    card:=word; (* this will not compile *)
    int :=word; (* this will not compile *)
    set :=word; (* this will not compile *)
    size:=SIZE(WORD) (* This equals 2 *) ...
```

LONGWORD

LONGWORD is a special type exactly the size of a 68000 long word (32 bits), thus it can hold values ranging from 0 to 4 294 967 295. No operations can be performed on values of type LONGWORD, except assignment. Any value which is held in memory within exactly 32 bits can be directly assigned to LONGWORD. This includes LONGCARD, LONGINT, REAL, ADDRESS, POINTERS, LONGBITSET and SETs. Variables of type LONGWORD must be word aligned in memory. This is a limitation of the 68000 processor. The compiler will normally ensure that this alignment is done.

LONGWORD will typically be used to create a parameter of type ARRAY OF LONGWORD in procedures. This allows the procedure to accept any type of parameter at least 32 bits in size and map the parameter into an array of words.

For example, LONGWORD can be used such as:

```
...
VAR
    lword : LONGWORD;
```

```

lcard : LONGCARD;
lint  : LONGINT;
lset   : LONGBITSET;
size   : CARDINAL;

```

BEGIN

```

lword:=lcard;
lword:=lint;
lword:=lset;
lcard:=lword; (* this will not compile *)
lint :=lword; (* this will not compile *)
lset :=lword; (* this will not compile *)
size:=SIZE(LONGWORD) (* This equals 4 *)
...

```

ADDRESS

The type ADDRESS is defined as a POINTER TO BYTE. The type is assignment compatible with any pointer type. For convenience, the type is also compatible with the LONGCARD type. This allows easy calculations to be performed to determine machine addresses. Other implementations of Modula-2 may have ADDRESS defined as a POINTER TO WORD, but since the BYTE is the smallest addressable unit on the 680x0 processor, it is essential to declare it as a pointer to a byte instead.

STRPTR

STRPTR is a special type used mostly by the Amiga interface modules. It is defined as a pointer to an array of characters. This means that when a parameter is of type STRPTR, the value expected should be the storage address of a string, or of an ARRAY OF CHAR.

NOTE: It is not recommended to pass a dereferenced STRPTR variable to a procedure expecting a value parameter of type ARRAY OF CHAR, unless the procedure uses the \$\$ compiler switch. Otherwise, the entire string pointed by the variable will be copied on the stack (since the pointer points at an array of a theoretical length of 32000 elements, all 32000 elements are copied on the stack).

PROCEDURE ADR(VAR object:AnyType): ADDRESS;

This procedure returns the storage address of a variable, a procedure, or a string literal. This will often be required when passing parameters to system procedures.

PROCEDURE TSIZE(AnyType): INTEGER;

This procedure returns the number of bytes of memory required to store one instance of the given type. This is invaluable information when trying to allocate an instance of the type dynamically. TSIZE() is retained for compatibility only, the standard SIZE() procedure should now be used instead.

PROCEDURE SHORT(long:LongType): ShortType;

This converts the given 32-bit value to it's 16-bit counterpart. This is done by discarding the upper 16 bits of the value.

PROCEDURE LONG(short:ShortType): LongType;**PROCEDURE LONG(hi,lo:ShortType): LongType;**

The first form of this procedure transforms the 16-bit input value into it's 32-bit counterpart, sign extending the value if necessary.

The second form of the procedure combines two 16-bits values to form one 32-bit value. The first parameter becomes the high 16 bits of the 32-bit value, while the second parameter becomes the low 16 bits.

PROCEDURE SHIFT(val:AnyType; int:INTEGER): AnyType;

This procedure performs an arithmetic shift operation on the bits composing the input value and returns the result. val is the value to shift, int is the number of bits to shift the value. A positive value indicates a shift to the left, a negative value a shift to the right.

PROCEDURE CODE(value:CARDINAL,...);

The CODE() procedure is used to directly include 680x0 instructions within a Modula-2 program. This procedure accepts an unlimited number of arguments. All arguments must be constants. The values will be included directly within the instruction stream at the given position. This is sometimes useful to include assembly language routines in a Modula-2 program.

PROCEDURE REGISTER(register:CARDINAL): ADDRESS;

This procedure reads the contents of a 680x0 processor register. You must specify a register to read. A value of 0 indicates register D0, a value of 7 indicates register D7, a value of 8 indicates register A0, and a value of 15 indicates A7. This translates to:

0..7 = D0..D7
8..15 = A0..A7

PROCEDURE SETREG(register:CARDINAL; val:AnyType);
This procedure sets the value of a specific 680x0 register. You must specify the register to affect. A value of 0 indicates register D0, a value of 7 indicates register D7, a value of 8 indicates register A0, and a value of 15 indicates A7. This translates to:

0..7 = D0..D7
8..15 = A0..A7

Regardless of the size of the parameter, the value deposited in the register will always be extended to 32 bits.

PROCEDURE LOADREGS(registers:BITSET);
For experienced programmers only.

You pass a set of registers respecting the same numbering conventions as for the REGISTER() procedure. The contents of the registers are loaded from the system stack using the 680x0 MOVEM.L instruction. This is useful in conjunction with the SAVEREGS() procedure defined below.

PROCEDURE SAVEREGS(registers:BITSET);
For experienced programmers only.

You pass a set of registers respecting the same numbering conventions as for the REGISTER() procedure. The contents of the specified registers are then saved on the system stack. They can later be reloaded using the LOADREGS() procedure defined above.

The SAVEREGS/LOADREGS() combination is useful in writing Amiga handlers. Handlers must preserve the CPU registers. To do so, you simply include a call to SAVEREGS() at the start of your procedure to preserve the values on the stack. The code in your procedure can then modify the contents of the registers as it executes. Before exiting the procedure, you include a call to LOADREGS() which fetches the original contents of the registers from the stack and adjusts the registers in accordance.

PROCEDURE GETA4();

For experienced programmers only.

This procedure will reload the current module's data pointer into the A4 680x0 register. A4 is used to reference global module data. This is identical in purpose to the `geta4()` procedure found in Amiga "C" compilers. There is an implicit `GETA4()` at the start of every procedure put there automatically by the compiler, unless the `$X` or `$L+` switches are used.

PROCEDURE TERMPROC(procedure:PROC);

This procedure implements an extremely useful extension to Modula-2.

Every Modula-2 module can have special initialization code located in the body of the module. This code gets executed automatically at the start of the program, before the main program starts. This is very useful; the initialization code can prepare resources used by the module and set the values of global variables.

Unfortunately, standard Modula-2 does not provide an equivalent mechanism for termination. The resources allocated in the module initialization code must eventually be released before the program terminates. In standard Modula-2, there has to be a special procedure defined which the main program must call. This procedure can then deallocate the module's resources. This approach works, but is detrimental to the abstraction facilities of the module concept, as it has to provide access to a procedure used strictly for internal processing.

`TERMPROC()` allows you to define a procedure which will get called automatically when your program exits. You simply pass `TERMPROC()` a procedure name, and that procedure will get called when your program exits.

There may only be one `TERMPROC()` installed for every module. Installing a second `TERMPROC()` removes the first one. Local modules cannot use `TERMPROC()`, it is a service only available for global modules. The installed `TERMPROC()` procedures are called in the reverse order of module initialization.

Standard Modula-2 Data Types

The following list describes all the standard Modula-2 data types available in M2Sprint. These types are used as building blocks to construct more sophisticated data types. This list does not include the data types available in the SYSTEM module, these are described in the previous section.

M2Sprint supports all standard Modula-2 data types, plus adds a few new types to take full advantage of the advanced 680x0 architecture of the Amiga.

SET

A collection of up to 32 elements. The number of bytes required to represent a set type depends on the range of the set.

- SET OF [0..7] requires one byte
- SET OF [0..15] requires two bytes
- SET OF [0..31] requires four bytes

Every element of a set is mapped (represented by) a single bit of computer memory.

enumeration

A list of up to 256 constants. A variable of this type always occupies one byte of storage.

ARRAY

Arrays can contain up to 32K of data. Arrays are always rounded to the nearest word boundary (this is forced by the 680x0 architecture).

RECORD

Records can contain up to 32K of data. Each 16- or 32-bit value in the record is padded to a word offset from the start of the record, and a complete record is always padded to the nearest word boundary (this is forced by the 680x0 architecture).

PROCEDURE

The procedure type acts as a pointer to a procedure's executable code. The type requires four bytes of storage.

POINTER

A pointer points to an element of a specified type. The size of a pointer is four bytes.

SUBRANGE

A subrange must be contained in the range [-32768..32767]. A subrange type always occupies two bytes of storage.

BOOLEAN

The boolean type requires one byte of storage. Zero represents FALSE, and all other values represent TRUE. This differs from standard Modula-2 implementations and is so to be compatible with the way the Amiga OS uses boolean values.

BITSET

BITSET is defined as SET OF [0..15] and occupies two bytes of storage. Every element of a set is mapped (represented by) one bit of computer memory.

CARDINAL

Can have a range of 0 to 65535 and occupies two bytes. It is assignment compatible with INTEGER, LONGINT and LONGCARD.

CHAR

Contains a value from 0 to 255 that represents an ASCII character. CHAR types occupy one byte of storage.

INTEGER

Can have a range of -32768 to 32767 and occupies two bytes of storage. It is assignment compatible with CARDINAL, LONGINT and LONGCARD.

LONGBITSET

LONGBITSET is defined as SET OF [0..31] and occupies four bytes of storage. Every element of a set is mapped (represented by) one bit of computer memory. LONGBITSET is not assignment compatible with BITSET. When referring to a LONGBITSET constant you need to prepend the identifier LONGBITSET to the constant. For example, {1,4,24} is not a valid LONGBITSET constant. It would have to be represented as LONGBITSET{1,4,24}.

LONGCARD

Can have a range of values from 0 to 4 294 967 295 and occupies four bytes of storage. It is assignment compatible with INTEGER, CARDINAL and LONGINT.

LONGINT

Can have a range of values from -2 147 483 648 to 2 147 483 647 and occupies 4 bytes of storage. It is assignment compatible with INTEGER, CARDINAL and LONGCARD.

LONGREAL

LONGREAL numbers are represented in the standard IEEE notation. M2Sprint uses the Amiga libraries for LONGREALs. This means that if a 68881 math coprocessor is installed, programs compiled under M2Sprint will automatically take advantage of it. LONGREALs require 8 bytes of storage and can be in the following range:

Positive Max: 1.797693134862316D+308

Positive Min: 2.225073858507201D-308

Negative Max: -1.797693134862316D+308

Negative Min: -2.225073858507201D-308

LONGREAL constants must be followed by a "D" to indicate they are LONGREAL instead of simple REAL.

PROC

Type compatible with procedures with no parameters and no return value. Uses four bytes of storage.

REAL

REAL numbers are represented in the Motorola Fast Floating Point

format. REALs occupy four bytes of storage and can be in the following range:

Positive Max: $9.223371\text{E}+18$

Positive Min: $5.421011\text{E}-20$

Negative Max: $-9.223371\text{E}+18$

Negative Min: $-5.421011\text{E}-20$

Standard Modula-2 Procedures

The following is a list of the standard pre-defined Modula-2 procedures available in M2Sprint. These procedures are considered pre-declared in the Modula-2 language, and thus do not need to be imported from any module and can be used directly in any part of a program.

This list of standard procedures includes the name of each procedure with an attempt to define its interface as a standard Modula-2 declaration. There are several references to `ScalarType`. This indicates that the given value can be any scalar type. Scalar types include `INTEGER`, `CARDINAL`, `LONGINT`, `LONGCARD`, `BOOLEAN`, `CHAR`, enumerations and subranges. Additionally, many procedures accept different types of parameters. In this case, all the different types accepted are listed, separated by slashes ("/").

Please refer to any good Modula-2 textbook for examples of how to use these standard procedures.

**ABS(value:INTEGER/LONGINT/REAL/LONGREAL):
INTEGER/LONGINT/REAL/LONGREAL;**

Return the absolute value of the input parameter

CAP(c:CHAR): CHAR;

Return the upper case version of the input character. This will transform the range of characters ["a".."z"] into ["A".."Z"], and leave all other characters unchanged. Note that `CAP()` does NOT handle the international character set as implemented on the Amiga, so it will not capitalize characters in the range [200C..237C].

CHR(x:ScalarType);

Return the character with ordinal value 'x'.

DEC(VAR x:ScalarType);

DEC(VAR x:ScalarType; amount:ScalarType);

Decrement a variable by one or 'amount'. This is equivalent to

`x := x - 1;`

or

`x := x - amount;`

The `DEC()` procedure generates better code than the above, and also improves source code readability.

EXCL(s:SetType; element:SetElement);

Exclude an element from a set. This is equivalent to

```
s:=s-SetType{element};
```

The EXCL() procedure generates better code than the above, and also improves source code readability.

FLOAT(x:INTEGER/CARDINAL/LONGINT): REAL;

Convert a value of type INTEGER, CARDINAL or LONGINT to REAL. Certain implementations of Modula-2 may restrict the conversions to CARDINAL values only.

FLOATD(x:INTEGER/CARDINAL/LONGINT): LONGREAL;

Convert a value of type INTEGER, CARDINAL or LONGINT to LONGREAL. Certain implementations of Modula-2 may restrict the conversions to CARDINAL values only.

HALT

Immediately stop program execution. This will cause all installed TERMPROCs to be called, just like on normal program termination. If the Debug module is imported in a program, when execution encounters the HALT statement, the user gets the option to activate the M2Sprint debugger. In this case, TERMPROCs are only executed after having called the debugger.

HIGH(array:ArrayType): ArrayIndexType;

Return the highest element in an array. This only works for open array parameters, or for normal arrays whose lower index bound is 0.

INC(VAR x:ScalarType);

INC(VAR x:ScalarType; amount:ScalarType);

Increment a variable by one or 'amount'. This is equivalent to

```
x:=x+1;
```

or

```
x:=x+amount;
```

The INC() procedure generates better code than the above, and also improves source code readability.

INCL(s:SetType; element:SetElement);

Include an element from a set. This is equivalent to

```
s:=s+SetType{element};
```


The `INCL()` procedure generates better code than the above, and also improves source code readability.

**`INLINE(lib:ADDRESS/CONST; offset:INTEGER;
register:CARDINAL, register...)`**

Define an Amiga library routine.

The Amiga uses a very efficient means of accessing system functions, the program simply needs to jump at an offset from a given address. There is no context switch or exception handling needed, thus providing a very low-overhead interface.

Most Amiga languages must provide stub routines to interface a program to the system libraries. The stub routines take the standard language parameters passed on the stack, puts them into CPU registers and jump to the system routines. This is not as efficient as it should be, both for code size and speed.

`M2Sprint` has a special feature which will automatically convert a normal procedure call into a special Amiga library call. The parameters are automatically put in CPU registers and a jump is made to the ROM code. This is both faster and smaller than stub routines.

`INLINE()` implements this feature and is obviously an extension to standard Modula-2, tailored especially to the Amiga. To declare a procedure as being an Amiga library procedure, the declaration of the procedure must be followed by a call to the `INLINE()` procedure. As such, most Amiga procedures defined in the standard `M2Sprint` Amiga modules have `INLINE()` statements following the procedure declarations. An important point is that `INLINE()` is mainly used in definition modules. It is the **ONLY** procedure that can be called from a definition module. This is an exception to standard Modula-2, and has been implemented this way for efficiency reasons.

`INLINE()` takes a variable number of parameters. The first parameter corresponds to a variable pointing to the library the procedure belongs to. For example, all procedures in the `Intuition` module use the variable `IntuitionBase` as first parameter.

The second parameter corresponds to the offset in the library of the procedure. This is always a negative value which is a multiple of 6 and starts at -30 (-30, -36, -42, etc...). These values must be obtained from the documentation accompanying the library.

The rest of the values indicate in which registers to put every parameter. The third `INLINE()` parameter indicates in which register to put the first parameter of the procedure, fourth `INLINE()` parameter indicates in which register to put the second parameter, etc... If the procedure has no parameters, the `INLINE()` procedure call only has two parameters (the library, and the procedure offset). Register numbers follow the same conventions as the `REGISTER()` procedure from the `SYSTEM` module (`D0=0..D7=7,A0=8..A7=15`);

Amiga library routines return their result values in the `D0` CPU register. If the procedure declaration specifies a return value, `M2Sprint` will use the value of `D0` at the time execution returns from the ROM call as the return value for the procedure call.

Here's an example of using `INLINE()` to declare a DOS library procedure:

```
PROCEDURE Open (name:STRPTR; accessMode:LONGINT) :  
                FileHandle;  
  INLINE (DOSBase, -30, 1, 2);
```

The first parameter of `INLINE()` is `DOSBase` which is a variable that points to the DOS library.

-30 is the offset within the library of the `Open()` function, as specified by the library documentation.

The two other values specify the registers to use to pass the parameters to the library procedure. In this case, the first parameter goes in register 1 (`1=D1`) and the second parameter goes in register 2 (`2=D2`). These register numbers must also be obtained from the documentation accompanying the library.

Finally, the DOS library's `Open()` function returns a value in `D0` which is automatically picked up by the compiler due to the fact that the procedure is declared as returning a value (`FileHandle`).

Note that when a procedure is declared as `INLINE`, it does not have an implementation part. So even though the procedure is defined in a definition module, it is not necessary to have it present in the implementation part. See the `$I` compiler switch for a related feature.

MAX(AnyType): AnyType;

Return the maximum value for any given type.

MIN(AnyType): AnyType;

Return the minimum value for any given type.

**ODD(x:INTEGER/CARDINAL/LONGINT/LONGCARD):
BOOLEAN;**

Return TRUE if the parameter is odd, or FALSE if it is even.

ORD(x:AnyType): CARDINAL;

Return the ordinal position for the parameter within its set of possible values, starting at 0.

SIZE(AnyType): CARDINAL

SIZE(x:AnyType): CARDINAL;

Return the number of bytes of storage required to store a specific type or a variable.

TRUNC(x:REAL/LONGREAL): LONGINT;

Return the integer component of a REAL or LONGREAL value.

TRUNCD(x:REAL/LONGREAL): LONGINT;

This procedure is an exact synonym of the TRUNC() procedure above. It is present in this implementation for compatibility only. Please use the TRUNC() procedure instead for any new code.

VAL(AnyType1; y:AnyType2): AnyType1;

Implements type transfer. Same as AnyType1(y).

Standard Modula-2 operators and delimiters

Operators and delimiters are the special characters, character pairs or reserved words listed below. The reserved words consist exclusively of capital letters and must not be used in the role of identifiers. The symbols # and <> are synonyms and so are &, AND, and ~, NOT.

+	=	AND	FOR	PROCEDURE
-	#	ARRAY	FORWARD	QUALIFIED
*	<	BEGIN	FROM	RECORD
/	>	BY	IF	REM
:=	<>	CASE	IMPLEMENTATION	REPEAT
&	<=	CONST	IMPORT	RETURN
.	>=	DEFINITION	IN	SET
,	..	DIV	LOOP	THEN
;	:	DO	MOD	TO
()	ELSE	MODULE	TYPE
[]	ELSIF	NOT	UNTIL
{	}	END	OF	VAR
^		EXIT	OR	WHILE
~	(space)	EXPORT	POINTER	WITH

Standard Modula-2 predefined identifiers

The following is a list of the standard predefined Modula-2 identifiers available in M2Sprint. These identifiers are considered pre-declared in the Modula-2 language, and thus do not need to be imported from any module and can be used directly in any part of a program.

ABS	FALSE	LONGBITSET	ORD
BITSET	FLOAT	LONGCARD	PROC
BOOLEAN	FLOATD	LONGINT	REAL
CAP	HALT	LONGREAL	SIZE
CARDINAL	HIGH	MAX	TRUE
CHAR	INC	MIN	TRUNC
CHR	INCL	NIL	TRUNCED
DEC	INLINE	ODD	VAL
EXCL	INTEGER		

Extensions

Although M2Sprint tries to conform as closely as possible to the official definition of the Modula-2 language, a few additions were made to make Modula-2 more useful on the Amiga. It is important to note that use of these features may make it difficult to port the resulting code to a different compiler.

The Modula-2 language definition specifies the syntax for an identifier as:

```
ident = letter {letter | digit}
```

M2Sprint extends the definition to:

```
ident = letter {letter | digit | "_"}
```

Thus the underscore character is now allowed in identifiers. For example, the following is perfectly valid in M2Sprint:

```
VAR
    my_Window : WindowPtr;
```

A special feature has been added to improve the handling of string literals. It is now possible to embed commands, much like in the "C" language, directly into string literals. Every command is introduced by a backslash character ("\") and is followed by one or more characters. The following commands are supported:

<code>\b</code>	Put a backspace (BS) character in the string.
<code>\f</code>	Put a form feed (FF) character in the string.
<code>\n</code>	Put a line feed (LF) character in the string.
<code>\r</code>	Put a return (CR) character in the string.
<code>\t</code>	Put a horizontal tab (HT) character in the string.
<code>\v</code>	Put a vertical tab (VT) character in the string.
<code>\xNN</code>	Put the character represented by the hexadecimal constant formed by NN in the string.
<code>\NNN</code>	Put the character represented by the octal constant formed by NNN in the string.

Example of usage:

<code>"string\n"</code>	gives: string<LF>
<code>"\fstring:nstring2"</code>	gives: <FF>string<LF>string2
<code>"\x41BCDE"</code>	gives: ABCDE
<code>"\101BCDE"</code>	gives: ABCDE

`"\\string"`

gives: `\\string`

When translating programs written under compilers not supporting the extended string literal facility, there may be some problems with string literals using the `"\"` character. As shown in the examples above, putting two `"\"` in a row causes the compiler to output a single `"\"`. As well character constants represented by `"\"` can be converted to the constant 134C.

NIL has a value of 0 in M2Sprint. This is identical to the NULL value in the "C" language. This is extremely convenient for programming the Amiga which requires a NIL pointer to be equal to 0.

The INLINE facility is provided to interface to the Amiga ROM libraries. Please refer to the previous discussion in this chapter for more information on INLINE.

Limitations

Although M2Sprint tries to conform to the standard Modula-2 definition, there are certain details which were done differently or are limited by implementation restrictions. The following list explains these limitations.

M2Sprint does not support the module priority concept. The compiler does accept them when reading a source file, but it discards the value once read.

The standard `HIGH()` procedure has intentionally been limited to manipulating either open array parameters or static arrays whose lower index bound is 0. Trying to do a `HIGH()` call on an array whose lower bound differs from 0 will cause a compilation error.

The previously standard `NEW()` and `DISPOSE()` procedures are no longer part of the Modula-2 language definition, and are thus not implemented in M2Sprint. The `ALLOCATE()` and `DEALLOCATE()` procedures from the Storage module may be used instead.

The `SIZE()` and `VAL()` procedures are available as predefined identifiers and do not need to be imported from `SYSTEM`. This conforms to the latest definition of the Modula-2 language. The `TSIZE()` procedure from the `SYSTEM` module should no longer be used, new code should try to use the `SIZE()` procedure instead.

`EXPORT` statements are ignored by the compiler. This conforms to the latest definition of the Modula-2 language. All identifiers declared in a definition module are now considered automatically exported.

Coroutines are no longer part of the standard Modula-2 language definition, so they are not directly supported by M2Sprint. Several modules are provided to utilize the much superior Amiga multitasking abilities instead (see the `TaskUtils` module for example).

All 16- and 32-bit values must be located on a word boundary in memory. This is a limitation of the 68000 processor. This alignment is

done automatically for you by the compiler in most cases. Problems can arise when using pointers though, you must ensure that pointers to 16- or 32-bit values have an even value.

Since M2Sprint is a single-pass compiler, the source text is processed sequentially from top to bottom. As such, any identifier referenced must have been declared textually before the reference. A relaxation of this rule applies to pointer types which may be declared as pointing to an object which has not yet been defined.

It is sometimes essential to be able to call a procedure which has not yet been defined, in the case of mutual recursion for example. To aid in this area, M2Sprint supports FORWARD procedures which serve to give a hint to the compiler concerning the types of parameters for an as-of-yet undefined procedure. The best way to illustrate the use of FORWARD procedures is through an example.

In this example, the procedure GetXPos() is positioned after its calling procedure CalcPos(). The compiler must be made aware of the existence of the GetXPos() procedure before CalcPos() tries to call it. This is needed to perform complete type-checking on all the parameters of the procedure call. To have the compiler understand the situation, the heading of the GetXPos() procedure must be positioned before the CalcPos() procedure within the module, and the keyword FORWARD must be added following the heading. Without the line declaring GetXPos() as FORWARD before CalcPos(), the compiler would refuse to compile this code fragment:

```
...
PROCEDURE GetXPos(): CARDINAL; FORWARD;
    (* notifies the compiler of
     * GetXPos
                                     *)
PROCEDURE CalcPos;
BEGIN
    x := GetXPos;
...
END CalcPos;

PROCEDURE GetXPos(): CARDINAL;
...
END GetXPos;
...
```

Size Limitations

Regardless of the theory behind a language, a functioning implementation must always impose some limitations, be it due to the underlying operating system, the CPU or the language itself. Most size limitations imposed by M2Sprint are very unlikely to be encountered:

Maximum executable code per module	: 32K
Maximum global variables per module	: 32K
Maximum string data per module	: 32K
Maximum array size	: 32K
Maximum record size	: 32K
Maximum executable code per program	: Unlimited
Maximum global variables per program	: Unlimited
Maximum string data per program	: Unlimited
Maximum number of imported modules per compilation unit	: 64
Maximum number of constants in enumeration types	: 256
Maximum number of EXIT statements in a LOOP construct	: 16
Maximum number of nested WITH constructs	: 4
Maximum number of cases in a CASE construct	: 128
Maximum number of elements in a SET	: 32
Maximum size of procedure return values	: 8 bytes

Executable Memory Usage

Executable code for an M2Sprint program is contained in `hunk_code`. Global variables are allocated when a program is started, it is contained within the executable as `hunk_bss`. Strings are held in a `hunk_data`.

All local variables are allocated on the stack.

Register Usage

This section describes the run-time usage of the CPU registers by an M2Sprint program. All "scratch" registers do not need to be preserved by a procedure call (including registers D0 and D1).

- D0 Contains the lower 32 bits of procedure return values
- D1 Contains the upper 32 bits of procedure return values
(for LONGREALs)
- D2 Scratch
- D3 Scratch
- D4 Scratch
- D5 Scratch
- D6 Scratch
- D7 Scratch

- A0 Scratch
- A1 Scratch
- A2 Scratch
- A3 Scratch
- A4 Pointer to current module's static data region
- A5 Stack frame pointer
- A6 Scratch (used for Amiga library calls)
- A7 Stack pointer

13. M2L - The Linker

M2L is a high speed single-pass program linker. Its job is to link together the object files that the compiler generates in order to produce an executable file which AmigaDOS can load and run as a stand-alone program.

In order to create an executable file, all the modules composing a program must be linked together. M2L handles this process and provides a few options to control how the executable will be generated.

Unlike older language systems, the linking process for a Modula-2 program is extremely simple. All that needs to be supplied to the linker is the name of the main program module. From that point, all modules which the main module relies on are fetched automatically.

Linker Options

The M2Sprint linker offers a few options which control how executable code is generated. These options can be controlled in various ways. This section describes the options themselves, following sections will describe how to adjust these options from the different environments available.

Run time support

The M2Sprint linker automatically links in a special module with every program that it generates. The module, known as "RunTime", contains all of the code necessary to support Modula-2 programs. This includes code to handle such tasks as program startup and termination, REAL/LONGREAL ROM interface routines and 32-bit INTEGER operations.

Although the overhead incurred by linking with RunTime is minimal, it still constitutes extra code appended to each and every program created. In order to reduce this overhead, several different versions of the RunTime module have been provided. A special naming convention is used to distinguish them.

All the names start with RT, and have one of 2 letters appended to them:

- A This version of RunTime includes special code to open and close the ARP library automatically. If the library cannot be opened, the program simply displays a message to the user and exits.
- R This version of RunTime includes special code to interface Modula-2 programs to the Amiga's ROM FFP and IEEE math routines. If you intend to use any REAL/LONGREAL operations in a program, you MUST link with a version of RunTime with this special code.

The various versions of RunTime provided are:

- RT Bare-bones version of RunTime. When linking with this version, your program may not perform any REAL/LONGREAL operations, and will not have any run-time error checking.
- RTA Same as RT, except that the ARP library is automatically opened for you.

RTAR	This version includes ARP support as well as REAL/LONGREAL support.
RTR	This version provides REAL/LONGREAL support.
RTNoWB	This version of RunTime is designed to support the implementation of AmigaDOS file handlers. It is exactly like RT, with one exception: it does not provide any code to handle a Workbench startup. Programs using this RunTime module will not work from Workbench.

Hunk control

Determines the number of hunks that the executable will be made up of.

All AmigaDOS programs are composed of one or more hunks of code and data. There is a certain amount of overhead associated with each hunk, therefore programs made up of many hunks tend to be larger and take longer to load.

Programs with fewer hunks are smaller and load faster. However since a single hunk may not be subdivided when loaded into memory, large sections of contiguous available memory must be available in order for loading to be successful. Dividing a program up into more hunks means that the system can utilize "scatter-loading" techniques to store the program in small isolated sections of memory.

The M2Sprint linker provides three different ways of organizing the various hunks composing a program. You can experiment with these options to figure out which one gives the best performance for the application you are working on:

- **One Hunk Per Program**
This will cause the linker to combine all hunks from the executable into one large hunk. This provides the smallest code possible, but also requires one contiguous region of memory to load the executable program. This is the best option for smaller applications.
- **One Hunk Per Module**
This will cause the linker to generate one hunk for every module that is linked in. This produces the most hunks possible. It is very good in order to create programs which load even

though there are no large chunks of free memory available, but it adds substantial overhead in program size and loading speed.

- **Kilobytes Per Hunk**

This option allows you to fine-tune how the executable will be laid out. You provide this option with an optimum hunk size in Kilobytes. The linker will then try to combine modules in such a way as to generate hunks of that size, or less. So if your program is around 100K in size, you can request that there be four modules created by setting this option to 25K. This is the best solution for large applications. A value between 15 and 30 is quite appropriate here.

Code generation options

These options affect how the linker produces code and what happens at code generation time.

Debugging Information

When this option is selected, the M2Sprint linker will include debugging information in the executables it generates. Although programs that are linked with this option ON will take up slightly more disk space, they will not use more memory when loaded as all the debugging information is automatically discarded. In order to use M2Debug or any other third-party symbolic debugger, the debug information must be included in the executable.

IMG File

This file contains the list of image files to be linked with the executable. The image files contain graphics that have been converted from the IFF format (as used by Amiga paint programs) to an object file format that the linker can understand. The conversion process from IFF format to object format is performed by the IFF2Obj utility. The object files can be used to add custom gadgets or extra graphics to your program easily. To access the graphic information linked to your programs, refer to the definition of the ChipData module in the "Library Reference Manual". For information on how to use this feature, see the section on IFF2Obj in Chapter 14 - "Support programs and utilities".

Linker Beep

The M2Sprint linker can beep whenever it has finished a link job. This is very useful when performing batch linking in the background while other work is being done.

Icons

The M2Sprint linker can generate icons for the files it produces. The linker uses the file "M2.prog.info" from the M2Data: drawer as a template when creating a new icon. That is, an exact duplicate of the "M2.prog.info" file is created under the name of the file the linker is generating.

Output Drawer

This option allows to specify in which drawer the linker is to put all of the files it generates. By default, all files are put in the current drawer.

Adjusting the default settings

The default configuration settings used by the M2Sprint linker are located in the file M2Data:M2.linkconfig. There are two ways to modify the contents of this file:

- by selecting the "Config/Linker Settings..." command from within the integrated environment.
- by using the LinkSettings tool from CLI or Workbench.

Please refer to Chapter 14 - "Support programs and utilities" for more information on how to use these requesters.

Linking from the environment:

1. Compile the program you wish to create.
2. Select the "Modula-2/Link" command.

If the current link job is the first, M2E must take the time to load the linker from disk. The linker is searched for as "M2L". If it is not found under that name, an attempt is made to load it as "M2Sprint:M2L". Once loaded, the linker will remain in memory and will not need to be reloaded for future link jobs. The memory used by the linker may be freed using the "Modula-2/Unload Comp/Link" command.

Linking from CLI:

1. Type "M2Sprint:M2L FileName" at a CLI prompt where FileName is the name of the Modula-2 file to be linked.

2. Press RETURN.

As all other M2Sprint tools, the M2Sprint linker can be run from CLI as a normal Amiga application. The following command-line template is supported by the linker:

```
FILES/... , TO/K, RUNTIME/K, HUNKS/K,  
IMG/K, DEBUG/S, FREQ/S, BEEP/S, HOT/S
```

This template can be obtained from CLI by starting the linker with a question mark ("?",) as only parameter. The template for the linker works exactly like normal AmigaDOS templates.

Any parameters passed to M2L from CLI will be used in place of the linker's settings as defined in M2Data:M2.linkconfig. If a switch such as "BEEP" is included in the command-line, it will toggle the value specified in M2.linkconfig. If BEEP is turned on in M2.linkconfig, including it in the command-line will turn the beeping off, and vice-versa. Here is a description of the various command-line options available:

FILES/...

Allows you to specify any number of files to link. Multiple filenames may be specified by separating each with a space. Filenames containing spaces must be surrounded by quotation marks. Filenames can include complete directory paths. When linking a program, it is not necessary to specify the .lnk extension, the linker will append it automatically.

TO/K

Allows you to specify a destination directory where the linker is to output executable files it creates. If this is not specified, the linker will output its files to the "Output Drawer" specified in the M2.linkconfig file. If the "Output Drawer" is empty, then output will go to the current directory.

RUNTIME/K

Allows you to specify the runtime module to use for this executable.

HUNKS/K

Allows you to specify the hunk layout of the executable. This option is implemented as follows:

0 means that there will be one hunk per program

1 means that there will be one hunk per module

All other values indicate a number of K for every hunk in the

executable.

IMG/K

Allows you to specify an IMG file to link in with the executable.

DEBUG/K

Allows you to invert the current setting of the "Include Debug Info" option.

FREQ/S

Allows you to invert the current setting of the "Use File Req" option.

BEEP/S

Allows you to invert the current setting of the "Beep After Compiling" option.

HOT/S

Allows you to start the linker as an ARexx server. Once started as a server, the linker can be called directly using ARexx commands. It can be aborted and unloaded from memory by hitting CTRL-C in the same CLI window where it was started from, or by using the AmigaDOS BREAK command. It can also be aborted by sending it the ARexx "QUIT" command. Refer to the section on linking from ARexx below for more information on the linker's ARexx interface.

Examples:

```
M2Sprint:M2L myProg hunks 0 beep
```

This will cause the linker to link the file "myProg.lnk", combining all the hunks together, and toggling the setting for the Beep option.

```
M2Sprint:M2L myProg1 myProg2 runtime RTA hot
```

This will cause the linker to link the files "myProg1.lnk" and then "myProg2.lnk" using the RTA runtime module. Once the link job finished, the linker will then enter ARexx server mode due to the presence of the HOT option.

Linking from Workbench:

Individual or multiple files can be selected for linking by shift-clicking their icons. Shift-double-clicking M2L's application icon will launch M2L and link each file selected in the order they were selected.

1. Find the file(s) to be linked.
2. Select the file(s) to be linked by clicking their icons while holding down the SHIFT key.
3. After the file(s) have been selected, double-click on the M2L icon while holding down the SHIFT key.

If the M2L icon is selected by itself, without shift-clicking any other files, the linker will open a window and present the command-line template described in the section "Linking from CLI" above. The linker will then behave exactly like if it were started from CLI. You can enter multiple filenames and enter switches.

Linking from an ARexx macro:

The M2Sprint linker can be used as an ARexx server. This allows it to be used with any ARexx-compatible text editor. The linker must be started with the HOT option explicitly specified (see "Linking from CLI" above) in order to use it as a server.

Once started in server mode, the linker opens a message port under the name "M2L_ARexx_Port" which can be used to send ARexx commands to it. Sending any of the configuration commands to the linker changes its internal defaults and will affect all following link jobs. A list of the supported ARexx commands and their respective parameters follows:

LINK filename

Sending this command to the linker will cause it to link the file <filename.link>. The filename can include a directory path. Note that the linker's current directory is the one from which it was started, and not the directory from which the ARexx command was invoked.

TO dirname

Allows you to specify a destination directory where the linker is to output all executables it creates. If this is not specified, the linker will output its files to the "Output Drawer" specified in the M2.linkconfig file. If the "Output Drawer" is empty, then output will go to the current directory.

RUNTIME name

Allows you to specify a version of the RunTime module to link with every executable.

HUNKS size

Allows you to specify the hunk layout of the executable. This option is implemented as follows:

0 means that all the modules in the program will be compined in one hunk

1 means that there will be one hunk per module

All other values indicate the number of K for every hunk in the executable

IMG name

Allows you to specify an IMG file to link in with the executable.

DEBUG [ON|OFF]

Allows you to adjust the current setting of the "Include Debug Info" option.

FREQ [ON|OFF]

Allows you to adjust the current setting of the "Use File Req" option.

BEEP [ON|OFF]

Allows you to adjust the current setting of the "Beep After Compiling" option.

Batch linking

When producing a series of programs, it is often useful to be able to link all the programs automatically. The M2Sprint linker provides a feature known as Batch Linking to assist in this area. This feature allows groups of Modula-2 link files to be linked quickly and easily.

A linker batch file consists of a list of filenames with optional parameters after each filename. As the linker processes the batch file it links each file listed.

Every line of a batch file should contain the name of a single link file. The line can also contain any linker options in the same format as if the linker was being launched from CLI.

NOTE: Comments can be included in the batch file by putting a semi-colon ";" at the start of a line, followed by the comment text.

To use Batch Linking from the environment, load the batch file in the editor, and select the "Modula-2/Batch Link" menu command. This will start the linking process.

To use Batch Linking from CLI, you must use IO redirection. For example, if the batch file is called batch.bat, you must start the linker from CLI using a command-line such as:

```
M2L <batch.bat
```

To use Batch Linking from Workbench, you must start the linker by itself by double-clicking its icon. You then use IO redirection as if it was started from CLI.

Batch linking cannot be invoked through the ARexx interface.

Locating link files

When linking a program, the linker will often require access to link files. Simply put, every module composing the program must be fetched.

Typically, the linker will look for link files in the current drawer, if not found, it will then look in the M2: drawer. If still not found, it will present a file requester prompting the user to select the file.

The M2Sprint linker supports a special feature that allows it to scan several different drawers for its link files. The file M2.searchpath found in the M2Data: drawer contains a list of all the drawers that should be searched. This is very similar in concept to the AmigaDOS PATH command.

As an example, if you have most of your link files in a RAM disk to speed up linking, but have not enough room to store all of them there. You could copy only the more used files to the ram disk and leave the less commonly referenced files on disk. You would then use M2.searchpath to tell the linker that certain files are on disk. M2.searchpath might look like:

```
RAM:MainM2  
DH0:ExtraM2
```

with a M2.searchpath file defined as above, the linker would first try to find link files by looking in the current drawer. If it's not there, then it would look in the M2: drawer. If still not found, then it would look in RAD:MainM2, and finally in DH0:ExtraM2.

Every line of the file indicates a specific search path. The path specification may be relative to the current drawer.

Linker output

After a succesful linking of a program, the linker reports certain information regarding the file it has just linked. Here is a list of the information which is displayed along with an explanation of every value:

Code

This value indicates exactly how many bytes of machine code the program just linked contains.

Global Data

This value indicates exactly how many bytes are required to hold all the global variables of the program.

String Data

This value indicates exactly how many bytes are used in the program to store strings.

Reloc Data

This value indicates how many bytes are used in the program to store relocation information. Relocation information is included in the disk version of an executable, but it is discarded when the program is executed, so it does not consume any memory.

Debug Data

This value indicates exactly how many bytes are used in the program to store debugging information. This value will be 0 if the "Include Debug Info" option is turned OFF.

Total

This value is simply the sum total of the previous five values. Note that this value does not correspond to the actual file size of the file output by the linker.

Using different versions of RunTime

The compiler does not see any difference between any the RunTime modules, as there is only ONE version of the definition module. Only the implementation of this module changes from version to version. For this reason, if your programs require some information from the RunTime module such as accessing any of the library bases, you should always import the variables from the module 'RunTime'. If you try to import something from 'RTAR' for example, the compiler will give an error.

You must provide the linker with the name of the implementation of RunTime your program requires. From the integrated environment, you specify the module name in the RunTime module string gadget of the "Config/Linker Settings..." requester. You can also pass the RunTime module name as parameter to the linker when it is run from CLI using the RUNTIME keyword. For example:

```
M2L myprog RUNTIME RTR
```

would cause the RTR version of RunTime to be linked with 'myprog'.

14. Support programs and utilities

M2Sprint's support programs and utilities are useful stand-alone applications that are designed to help programmers in a variety of areas.

Several programs such as M2FastLoad and M2A, have been created for the purpose of decreasing setup time overhead due to long load times.

Others, such as M2Prof and M2Debug, can actually help you write better Modula-2 programs.

NOTE: Although examples for using these utilities are only given for use from CLI, all of these programs can be used from Workbench also. By default, these programs reside in the M2Sprint: directory.

NAME

CompSettings - allows the default settings for the compiler to be adjusted

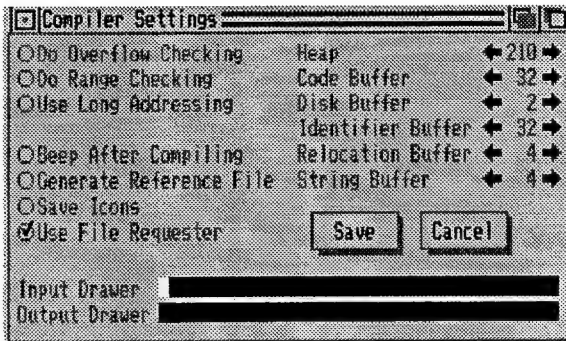
TEMPLATE

CompSettings

DESCRIPTION

Allows M2Sprint's default compiler switches and options to be adjusted and saved for later use by the compiler. Use the mouse to adjust the switches and values to their desired setting.

When "Save" is selected, the options are saved in the M2Data:M2.compconfig file and the next compilation will use these options,. Any of the options may be overridden by using command-line switches or through ARexx commands.



SEE ALSO

For more information on compiler switches and options see Chapter 12 "M2C - The Compiler".

NAME

IFF2Obj - convert IFF graphics images to object file format

TEMPLATE

IFF2Obj COMMAND_FILE/A,TO/K

USAGE

IFF2Obj <Command File> [TO output file]

Command File Text file containing the names of the IFF graphic files to be processed.

TO output file Optional filename for the file containing the converted graphics to be saved to. If this parameter is not supplied, IFF2Obj will save the object file as the <Command File>.o

DESCRIPTION

When writing Amiga applications, it is often necessary to include graphic imagery directly in a program, so it can be displayed to the user when the application is being used. Titles, menu items and gadgets are all cases that often require graphics.

An important point to consider when doing graphics on an Amiga is that all graphic-related data must reside in a special zone of memory called "CHIP" memory. It is called CHIP memory due to the fact that it is the region of memory that is accessible by the Amiga's custom chips. This means that any graphic that an application wants to display must somehow be located in CHIP ram.

The Amiga market has several powerful paint packages available, all of them save their files in IFF format. It seems only natural to want to take the output from these paint programs and use it directly for your graphics imagery in your programs. This is where IFF2Obj steps in.

IFF2Obj converts a series of IFF files into a format that can be understood by the M2Sprint linker's special IMG feature. An application can thus link in various IFF files and use them from the application program quite easily. IFF2Obj ensures that all the necessary data goes to CHIP memory.

The steps necessary to link and use IFF files in an application may be confusing at first, but the process will become easy after a little experimenting.

To use IFF2Obj:

1. **Create a file containing the names of all the IFF files that you wish to link with your application.**
2. **Type "IFF2Obj Filename" and press RETURN.**

"Filename" is the name of the command file produced in step number 1. IFF2Obj will read, convert, and merge each of the IFF files into one large linkable file with a .O extension.

By itself, the file produced is not of very great use. It needs to actually be made a part of the application program. This is done at the time the application program is linked.

How to link graphics to an application:

3. **Create a file containing the names of all the object files you wish to link with your application. Object files are the files output from IFF2Obj. Most likely, you'll only have one file to list at this stage.**
4. **Specify the name of the file created in step 3 above in the "IMG File" string gadget of the Linker Settings requester.**
5. **Link the application.**

Now at this point, the IFF data is linked to your application, but you still need to be able to access the data that gets linked. If you could not access it, it would be there, attached to your program, but you couldn't do anything with it.

Accessing graphic data from an application:

6. **The ChipData module is provided to allow you to access the data you have linked in at step 5 above. Specifically, the variable `dataPtr` is a pointer to an array of Intuition Image records holding the graphic data.**

EXAMPLE

The best way to illustrate the use of IFF2Obj is to provide a complete example.

Assume you have an application program called "MyProg" which requires 3 graphics files to be linked in: circle.pic, square.pic and triangle.pic.

First step is to create a list of all the graphics we wish to link in. In our example, the file would look like:

```
circle.pic  
square.pic  
triangle.pic
```

and would be saved under the name "IFFList".

Second step is to run the IFF2Obj program. This would be done like so:

```
IFF2Obj  IFFList TO Images.o
```

Next step is to create a file which contains the names of all the object files we wish to link with our application. In this case, we only have one object file: Images.o. So the file would look like:

```
Images.o
```

and would be saved under the name IMGList

Next step is to specify the name of the file created in the previous step in the IMG file gadget of the Linker Settings requester. In our case, we would need to enter:

```
IMGList
```

We then simply link the program in the usual way.

At this point, we have an executable, with graphics attached. These graphics must be accessed from the application. As mentioned above, the ChipData module is provided for this purpose. The dataPtr variable points to an array of Image records. Each slot in the array corresponds to one single IFF file. The graphics are organized in the array in the order they were linked in. So in our case:

```
dataPtr^[0]
```

holds the image for circle.pic

```
dataPtr^[1]
```

holds the image for square.pic

```
dataPtr^[2]
```

holds the image for triangle.pic.

The application can use these Image records as any other. So for example, they could be displayed using the Intuition DrawImage() procedure or by attaching them to gadgets or menu items.

SEE ALSO

Chapter 13 "M2L - The Linker"

NAME

LinkSettings - allows the default settings for the linker to be adjusted

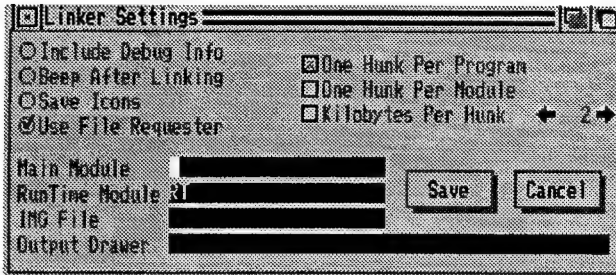
TEMPLATE

LinkSettings

DESCRIPTION

Allows M2Sprint's default linker switches and options to be adjusted and saved for later use by the linker. Use the mouse to adjust the switches and values to their desired setting.

When "Save" is selected, the options are saved in the M2Data:M2.linkconfig file and the next link job will use these options,. Any of the options may be overridden by using command-line switches or through ARexx commands.

**SEE ALSO**

For more information on compiler switches and options see Chapter 13 "M2L - The Linker".

NAME

M2A - used from Workbench or CLI to load existing files or open a new editor window while M2E is resident

TEMPLATE

M2A FILES/...,HOT/S

USAGE

M2A [<wildcards>] [HOT]

wildcards	Pathname, including wildcards, of existing files to be loaded for editing.
HOT	When this option is specified, it will cause M2A to only load the M2Sprint editor, without opening an editing window. The editor can then be invoked using the special hot key commands, or by invoking M2A once again without the HOT option.

DESCRIPTION

M2A can be run from Workbench or CLI to open a new window or load existing files from disk. M2A is much smaller than M2E so provides a way to load files without the time it takes to launch M2E.

If M2E is not already loaded in memory, M2A will load it from disk for you. It first searches for the editor as "M2E", and if it doesn't find it, it looks for "M2Sprint:M2E".

SEE ALSO

For more information on launching M2E and loading files into the editor read Chapter 11 - "M2E - The editor".

NAME

M2Batch - automates the process of creating compiler batch files

TEMPLATE

M2Batch MODULE/A,TO

USAGE

M2Batch <Modula-2 program source> [TO output file]

Modula-2 program source	Filename for the source code for which the batch file will be created.
TO output file	Batch file name. If no output file is specified the batch file will be saved using the same source filename with the extension ".bat".

DESCRIPTION

The M2Batch utility automates the process of creating a compiler batch file. By analyzing the IMPORT statements of a Modula-2 source file, M2Batch generates a compiler batch file containing the list of files the program is dependent on, in the order they need to be compiled.

In order to create a complete batch file with M2Batch, all of the Modula-2 source files required for a particular program must be present in the current directory. If M2Batch cannot find the source file for an imported module, it omits that file from the list. In short, M2Batch will organize the list as best as it is able given the available files.

M2Batch creates the output file under the name <Modula-2 program source>.bat where <Modula-2 program source> is the name of the source file being analyzed.

To use M2Batch:

1. Ensure that the necessary source files are in the current directory.
2. Type "M2Batch ProgName" where ProgName is the name of the main module.
3. Press RETURN.

M2Batch reads all available source files and creates a file that can be used with the M2Sprint compiler's Batch Compilation

feature.

SEE ALSO

Chapter 12 "M2C - The Compiler".

NAME

M2Debug - The post-mortem source-level debugger.

TEMPLATE

N/A

USAGE

N/A

DESCRIPTION**Overview**

M2Debug is a post-mortem source-level debugger. It is used to pinpoint the exact source statement which caused a program to crash.

On the Amiga, when a program misbehaves, the result is often a visit from the GURU. The GURU display offers only limited information as to the reason why an application crashed. The left number indicates the nature of the problem (see Alerts.def) and the right number indicates the address of the task which was active at the time of the crash.

System crashes can be caused by many things: referencing a NIL pointer, division by 0, poking random memory, etc... It is often very difficult to find exactly where in a program such problems occur. The result is a series of frustrating system crashes until the problem is found and eliminated.

M2Debug is a special tool which comes in to play after a program has crashed. When a Modula-2 program crashes, the user gets the choice to ignore the error and quit the program, or to load M2Debug and start a debugging session. In either case, the visit from the GURU is cancelled. If the user opts to load the debugger, M2Debug will display the exact Modula-2 code statement which caused the program to crash and will show the general state of the program prior to the crash, including the value of all variables, and the series of procedure calls which lead to the crash.

General concepts

Although M2Debug is simple to use, it may seem confusing unless the underlying concepts of its architecture are explained.

M2Debug's purpose is to show the state of a program at the time of a crash. This means showing the last Modula-2 statement to be executed before the crash, the value of all variables and the series of procedure calls which lead to the crash.

To accomplish this purpose, M2Debug maintains four types of information:

- Program source
- List of modules in the program
- Procedure chain; ie the series of procedure calls that lead to the crash
- The values of all global and active local variables

The program source is the ASCII text composing all the modules from a program. At the time of a crash, M2Debug attempts to load the source code to the crashed program and display the exact source statement which caused the crash.

The list of modules is a simple enumeration of all the modules composing the crashed program. M2Debug allows you to select any of these modules and see the related source code as well as viewing the values of that module's global variables.

The procedure chain is the list of the nested procedure calls which lead to the program crash. For example, if procedure A calls procedure B, procedure B calls procedure C, and procedure C crashes, then the procedure chain for this crash will be A, B and C.

After a crash, M2Debug maintains the memory used by the crashed program. This allows M2Debug to display the values of the program's variables, both global and local. Any global variable from any of the modules included in the module list may be displayed. Any local variable belonging to a procedure included in the procedure chain may also be displayed.

Knowing the values of all variables and the series of procedure calls which lead to a crash is extremely useful information in debugging applications. M2Debug's use goes beyond this however, in that it can be used to simply examine the value of all variables regardless if a crash occurred or not, using the Modula-2 HALT statement (see below). This is extremely useful when trying to find logic errors in

programs which do not necessarily cause system crashes.

Preparing to use M2Debug

To use M2Debug on a program, the program must have been properly prepared:

- The program must import the module 'Debug' at some point. The best place to do this is in the main module of the program by simply adding a line such as 'IMPORT Debug' in the list of import statements for the module.
- Modules should be compiled with the "Generate Reference File" option ON.
- The program must be linked with the "Include Debug Info" option ON.

When all the necessary steps have been taken, the resulting program may be executed. As soon as a crash situation is encountered, the program will display a requester explaining the nature of the crash, and will offer the choice to the user to either debug the program, or exit immediately.

M2Debug cannot be run as a stand-alone application. It may only be started after a program crash. Additionally, to run the debugger on a program, said program must have been started from CLI, and not from Workbench. From the integrated environment, this can be ensured by setting the appropriate flag in the "Config/Run Prog Settings..." requester.

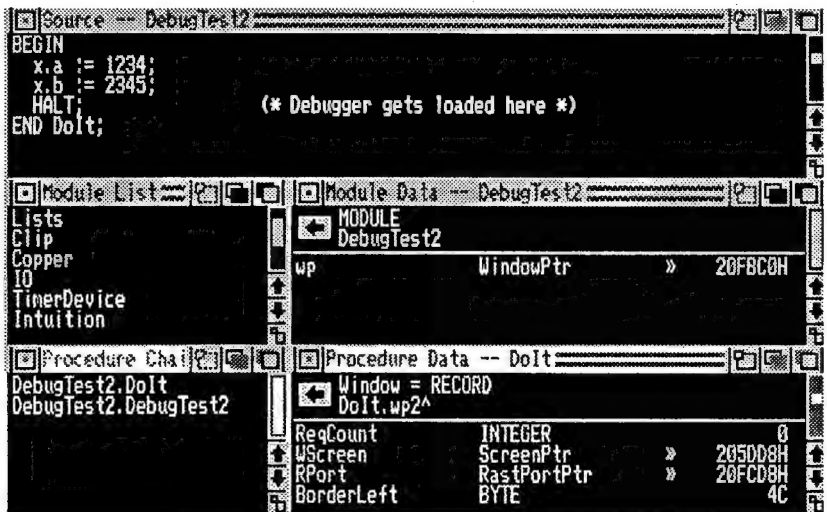
Extra help in debugging programs are the \$R and \$O compiler switches. Whenever these switches are turned on, the program will intentionally crash whenever either a range checking error or an overflow error are detected. This allows M2Debug to be loaded even in situations which do not necessarily constitute system crashes, but still are bugs in the program.

The Modula-2 HALT procedure can also be used to bring up the debugger at any point in a program's execution. When HALT is encountered, the normal events occur: the user can either debug or abort the program. If Abort is chosen, then the program simply exits. If Debug is chosen, then the debugger will be loaded in the normal fashion. After the debugger has quit, the user gets an additional

option of either resuming execution or aborting the program. This allows the user to include HALT statements in a program for the sole purpose of loading the debugger, and looking at a program's variables, etc.. and continue normal execution afterwards.

Using M2Debug

Once the 'Debug' option is selected from the requester which appears at the time of a crash, M2Debug will be loaded from disk and started automatically. The debugger is first searched for as 'M2Debug', and if not found, an attempt is made to load it as 'M2Sprint:M2Debug'.



When the debugger starts up, it opens a custom screen, and puts up five windows on it. The five windows are:

- **Source** - Displays the source code of the current module. This will initially show the program statement which caused the crash. The name of the currently loaded source file is displayed in the window's title bar.
- **Module List** - Displays the list of modules in the crashed program. This list includes only the modules which were compiled with the "Generate Debug Info" option turned ON.
- **Module Data** - Displays the value of all the global variables

defined for the currently selected module. The name of the currently selected module is displayed in the window's title bar.

- **Procedure Chain** - Displays the series of procedures that were called which lead to the crash. The top procedure in the list is the one in which the crash occurred.
- **Procedure Data** - Displays the list of variables local to the currently selected procedure. The name of the currently selected procedure is displayed in the window's title bar.

Each window can be closed by using the close gadget at the top left of each window. When all windows are closed, the debugger exits. The Reduce gadget at the top right of every window works differently than the one in the M2Sprint editor. When clicked, the gadget causes the window to toggle size from a large size to a small size and back. The scroll bar and scroll arrows on the right of every window can be used to move through the currently displayed information.

M2Debug has a menu strip at the top accessed by using the mouse menu button. The strip contains one menu with the following items:

- An item for every window. You can select these items to either open, or close specific windows.
- The 'Save Config' item causes the debugger to save the current screen layout (window sizes and positions) to disk. The next time the debugger is brought up, all the windows will be opened and disposed in the same format. The information is saved in the 'M2Data:M2.debugconfig' file.
- The 'About...' item displays the version number of the program.
- The 'Quit' item allows you to exit the debugger.

The source window

The Source window is used to display Modula-2 source code.

There are three ways to cause a specific module to be displayed:

- When the debugger is first started, it will display the source code of the module which contains the procedure which caused the crash.

- By selecting a module from the Module List window.
- By selecting a procedure from the Procedure Chain window.

When the user makes a request to load a new module, the debugger attempts to load the related source file from the current directory. If the file is not found, then a file requester is opened prompting the user to select the correct file. If 'Cancel' is selected from the file requester, the debugger will display '>> Source File Not Found <<' in the Source window. The name of the currently loaded module is displayed in the Source window's title bar in the format 'Source - <ModuleName>'.

The Module List window

This window presents a list of modules composing the crashed program.

The Module List window is used to select new modules to investigate. When a module is selected, the related source file will be loaded and displayed in the Source window, and the module's global variables will be displayed in the Module Data window.

To select a module, simply click the left mouse button over the module name in the window. When this happens, the debugger will attempt to load the module's source file. It will then attempt to load that module's Reference file. A module has a Reference file only if it was compiled with the "Generate Reference File" option turned ON. If a module does not have a reference file, then the values of its global variables cannot be displayed, and the Module Data window will indicate '>> Reference File Not Found <<'

The Procedure Chain window

This window presents a list of the procedure calls which lead to the program crash. The procedures are organized in the order in which they were called. The last procedure called (the procedure which crashed) is on the top of the list. The format of the display is:

```
ModuleName.ProcedureName
```

where ModuleName is the name of the module containing the proce-

cedure, and ProcedureName is the name of the procedure in this module.

The Procedure Chain window is used to select new procedures to investigate. When a procedure is selected, the source file for the module containing the procedure is loaded and displayed in the Source window, and the procedure's local variables are displayed in the Procedure Data window.

To select a procedure, simply click the left mouse button over the procedure name in the window. When this happens, the debugger will attempt to load the source file for the module containing the procedure. It will then attempt to load the Reference file for the module containing the procedure. A module has a Reference file only if it was compiled with the "Generate Reference File" option turned ON. If a module does not have a reference file, then the values of the local variables of the selected procedure will not be displayed, and the Procedure Data window will indicate '>> Reference File Not Found <<'

When selecting a new procedure, the Source window will highlight a particular statement in it's source listing. The highlighted statement corresponds to the link in the procedure chain. If the procedure chain looks like:

```
CalcMenus.HandleMenus  
Calc.DispatchEvents  
Calc.Calc
```

and the DispatchEvents procedure is selected, then the highlighted statement will be the call to procedure HandleMenus. If the HandleMenus procedure is selected instead, then the highlighted procedure will correspond to the last Modula-2 statement executed prior to the program crash.

The Module Data window

This window presents the list of all global variables from the currently selected module along with their values.

There are two ways to cause a module's global data to be displayed:

- When the debugger is first started, it will display the module data of the module which contains the procedure which caused

the crash.

- By selecting a module from the Module List window.

When the debugger attempts to load a new module, it will try to locate the module's Reference file. A module has a Reference file only if it was compiled with the "Generate Reference File" option turned ON. If a module does not have a reference file, then the values of its global variables will not be displayed, and the Module Data window will indicate '>> Reference File Not Found <<'. The name of the currently loaded module is displayed in the Module Data window's title bar in the format 'Module Data - <ModuleName>'.

The Module Data window contains three basic areas:

- The Data specification area
- The "Back Level" arrow
- The actual global data for the module

The Data Specification area presents two lines of text. The first line indicates the type of the information currently displayed in the Data area. The type of information will vary depending on the selection made within the data area. The second line of information is the name of the currently displayed information, again affected by the selections made in the data area.

To better illustrate the purpose of these two lines of text, it is best to use an example. If the data region is currently displaying the contents of a RastPort record obtained from a pointer in a Window record, the two lines of information would be:

```
RastPort = RECORD      ModuleName.window^.RPort^
```

The first line means that the currently displayed information corresponds to a RastPort, and a RastPort is a RECORD. The second line shows the origin of the information relative to your program. ModuleName is the module containing the variable, 'window' is the variable itself, and RPort is a field in the 'window' record variable.

The Back Level arrow allows you to go back from the currently displayed information to the previous level. Taking the above example again, hitting the Back Level gadget would yield:

```
Window = RECORD ModuleName.window^
```

The data region will now show the Window record corresponding to the window variable.

Finally, the Data region is where all the most important information is displayed. The information is organized in three columns:

- The first column indicates the names of variables or fields in a RECORD
- The second column indicates the type of the variable or field
- The third column indicates the value of the variable or record field

When the type of the variable is a pointer, a record or an array then a '»' symbol is displayed to indicate there is more information available. Clicking on a line with a '»' causes the debugger to move down one level and will display the extra information. When displaying nested information, the Back Level gadget can be used to back up one level. Again taking our example from above, if the Data Specification window indicates:

```
Window = RECORD ModuleName.window^
```

Then the data region will have several lines, one of which being:

```
RPort      RastPortPtr  »   212385
```

'RPort' is a field in the Window record. 'RastPortPtr' is the type of the RPort field. '»' indicates that there is additional information available. And '212385' represents the value of the field in hexadecimal notation.

Clicking on the RPort line will cause the Data Specification area to become:

```
RastPort = RECORD   ModuleName.window^.RPort^
```

and the data region will display the contents of the RastPort record. Again, simply clicking on the Back Level gadget will revert back to the level of the Window record.

Using the Module Data window, you can display contents of large records, of arrays and of simple variables. The best way to understand the functioning of the Data windows is to try debugging a sam-

ple program.

The Procedure Data window

This window presents the list of all local variables from the currently selected procedure along with their values.

There are two ways to cause a procedure's local data to be displayed:

- When the debugger is first started, it will display the local data of the procedure which caused the crash.
- By selecting a procedure from the Procedure Chain window.

When the user selects a procedure which is not in the currently selected module, the debugger attempts to load the new module's source and Reference file. A module has a Reference file only if it was compiled with the "Generate Reference File" option turned ON. If a module does not have a reference file, then the values of the selected procedure's local variables will not be displayed, and the Procedure Data window will indicate '>> Reference File Not Found <<'. The name of the currently selected procedure is displayed in the Procedure Data window's title bar in the format 'Procedure Data - <ProcedureName>'.

Please refer to the discussion of the Module Data window above to learn how to maneuver through the data display.

Keyboard Support

All menus items used by the debugger provide keyboard shortcuts. Additionally, the following keyboard equivalents are provided:

F10	= Same as clicking the zoom gadget
Up Arrow	= Same as clicking the up arrow gadget
Down Arrow	= Same as clicking the down arrow gadget
Left Arrow	= Same as clicking the Back Level gadget

Example of using M2Debug

To try out the debugger, type in the following program:

```

MODULE DebugTest;

IMPORT Debug;

VAR
    a,b,c : CARDINAL;

BEGIN
    a:=0;
    b:=30;
    c:=b DIV a;
END DebugTest.

```

After typing in this program, save it in the current directory.

When compiling this program, the "Generate Reference File" option should be turned ON. When linking this program, the "Include Debug Info" option should also be ON.

When you run this program, it will obviously result in a division by zero. You will get a requester stating that this is a run-time error due to a division by zero. You then have the choice to either click on Debug or Abort. Clicking 'Debug' will load the debugger and start it up.

The Source window will display the source code to the DebugTest program. The statement `c:=b DIV a;` will be highlighted to indicate that it was this statement which caused the crash.

The Module List window will show DebugTest, Lists, Debug and a couple of other modules.

The Module Data window will show the a b and c variables from the DebugTest module with their values.

The Procedure Chain window will be empty as will be the Procedure Data window. This is because the crash did not occur from within a procedure, but from within the body of the main module.

See the DebugDemo drawer on the M2Sprint disk for some example programs to test the debugger.

NAME

M2Errors - displays the errors from the last compilation

TEMPLATE

M2Errors W=WINDOW/S,NL=NOLISTING/S,NP=NOPAUSE/S

USAGE

M2Errors [WINDOW] [NOLISTING] [NOPAUSE]

WINDOW Redirects program output from the CLI window to a new window.

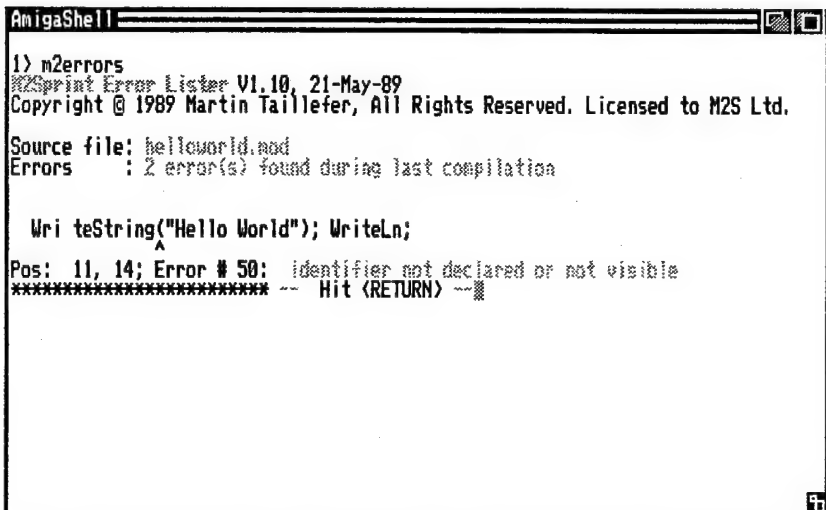
NOLISTING Does not display the offending line of source code. Only shows the error number and the line and column where the error occurs in the source code.

NOPAUSE Does not wait for a RETURN after each error.

DESCRIPTION

M2Errors interprets the "T:M2.errorlog" file generated by the compiler when it encounters compilation errors.

As the example below shows, errors are shown with the row and column position, error number, and error description. If the Modula-2 source file can be found, the offending line of code is also shown.



```
AmigaShell
1) m2errors
M2Sprint Error Lister V1.10, 21-May-89
Copyright © 1989 Martin Taillefer, All Rights Reserved. Licensed to M2S Ltd.

Source file: helloworld.mod
Errors      : 2 error(s) found during last compilation

Wri teString("Hello World"); WriteLn;
Pos: 11, 14; Error # 50: Identifier not declared or not visible
***** -- Hit <RETURN> --
```

After an error has been displayed, M2Errors waits for the user to press the RETURN key before going on to the next error.

By default M2Errors displays the output information in the CLI window from which it was launched. When run from Workbench, M2Errors automatically opens a window to display its information.

NAME

M2FastLoad - speeds loading of a group of files

TEMPLATE

M2FastLoad SOURCE/A,TO/A,MAKE/S

USAGE

M2FastLoad <fast file> [TO] <destination directory>

Reads a fastload format file.

fast file	Fastload format file to be read and unarchived.
destination directory	Directory to place the files extracted from the fastload file.

M2FastLoad MAKE <wildcards> [TO] <fast file>

Creates a fastload format file.

MAKE	Special "MAKE" switch to indicate M2FastLoad is creating rather than reading a fastload file.
wildcards	Pathname, including wildcards, for files to be included into the archive.
fast file	Name of the fastload file to be created.

DESCRIPTION

M2FastLoad is designed to speed the process of moving a large number of files to the RAM disk. Using M2FastLoad over AmigaDOS' COPY command can shorten loading time by as much as 75%.

M2FastLoad reads special "fastload" archives, in which any number of files can be stored in a compressed format. These files are decompressed in real-time and the contents placed in an output directory.

In order to take advantage of the program's capabilities, a fastload file must first be created with the program's MAKE option. The user specified files are compressed and organized into one fastload file which can then be loaded at any time.

To MAKE a fastload archive from CLI:

-
1. Locate all the files to be included in the fastload file.
 2. At the CLI prompt type "M2FastLoad Wildcard TO FastFile".

In this instance "Wildcard" is the pathname used to locate the source files. AmigaDOS and ARP compatible patterns can be mixed into the pathname to include or exclude certain filenames. FastFile is the name of the fastfile to be created.

3. Press RETURN.

M2FastLoad displays the name of each file as it is added to the fastload archive.

To extract files from a fastload archive:

1. At the CLI prompt type "M2FastLoad FastFile TO Directory".

Where FastFile is the fastload file to be read, and Directory is the destination directory where the extracted files are to be placed.

2. Press RETURN.

NOTE: A fastload archive containing all of the M2Sprint library modules has been provided on the disk entitled "Fastload Library". Those with enough memory should use M2FastLoad and the M2SprintLibs archive to speed the process of moving the libraries to the RAM disk.

EXAMPLE

M2FastLoad MAKE M2:#? to M2SprintLibs

This would cause all files from the M2: directory to be combined into the file "M2SprintLibs" in the current directory

M2FastLoad M2SprintLibs to M2:

This would extract all the files from the M2SprintLibs file and put them in the M2: directory.

NAME

M2Prof - helps programmers to optimize their code by use of a program profile

TEMPLATE

M2Prof PROGRAM/A,ARGUMENTS/...,TO/K

USAGE

M2Prof <M2 executable> [M2 executable arguments]
[TO output file]

M2 executable Name of program to be profiled.

M2 executable arguments Command-line arguments to be passed to the program being profiled.

TO output file Filename for the output to be sent to.

DESCRIPTION

M2Prof is designed to help programmers in the process of optimizing their code by use of a program profile.

M2Prof works by keeping track of the number of times each procedure in a program is called as well as the time spent inside each procedure. After the program has terminated execution, all of the statistical information is compiled into a table and saved in an ASCII file.

From this table programmers can easily see where the most time is spent during program execution. The program can then be rewritten to speed the operation of heavily used procedures, or restructured so that the use of slower procedures is minimized.

NOTE: Programs being profiled may NOT import the Debug module, as it interferes with the profiler.

To use M2Prof:

-
1. Type "M2Prof Executable" from CLI where **executable** is the name of a program that has been linked with the "Include Debug Info" option turned ON.
 2. Press RETURN.

By default, M2Prof writes the profile information to the filename <M2 Executable>.prf where <M2 Executable> is the name of the executable being profiled. The output can be redirected by specifying an output file before starting M2Prof.

The following example shows the result of a profile done on a program called "NameDraw". It uses the Random module and several simple procedures to simulate the drawing of names out of a hat.

From the table one can quickly see where the majority of time is being spent. In this case the InOut and Random modules undergo the heaviest usage. This indicates that if effort was applied towards improving procedures in those busy modules, the program's performance should increase.

Please note that although one of the goals in programming is to create the fastest code possible, one must always keep in mind the amount of time it will take to optimize compared to the gains to be had. Thus optimizing seldom used portions of a program will generally yield poor results.

NOTE: M2Prof classifies code that it could not understand properly as "unlabelled procedure".

EXAMPLE

M2Sprint Code Profiler Output
Copyright 1989 M2S Ltd, All Rights Reserved.

MODULE/PROCEDURE	# Calls	Time (secs)
MODULE NameDraw:		
Choose	18	0.10
DefineNames	2	0.00
Mainloop	2	0.00
PickNames	2	0.00
PrintNames	1	0.00
MODULE InOut:		
Write	14	0.38
WriteLn	14	0.00
WriteString	30	0.26
MODULE RandomNumbers:		
Random	488	0.45
MODULE Strings:		
LengthStr	30	0.00

NAME

M2XRef - provides a cross reference listing of elements from a given set of modules

TEMPLATE

M2XRef FILES/A,TO/A

USAGE

M2XRef <wildcards> [TO] <xref file>

wildcards Filename with optional wildcards that describe the files to be cross referenced (.sym files only).

xref file Name of the file where M2XRef is to send the cross referenced listing to.

DESCRIPTION

M2XRef provides you with the ability to obtain a complete cross reference listing of every element in a group of files. After analyzing the symbol files specified, M2XRef will create a three column table displaying the name of the element, the module from which it originated, and its type.

This utility can be especially useful when program maintenance is an issue. Creating a cross reference listing of a section of code can significantly ease the job of locating and tracking all the various elements of a program.

The entire M2Sprint library has been cross referenced and included in the appendix of the Library Module Reference.

NAME

MakeErrorList - allows compilation error descriptions to be customized

TEMPLATE

MakeErrorList INFILE/A,OUTFILE/A

USAGE

MakeErrorList <input file> <output file>

input file ASCII file containing the error codes and their descriptions.

output file Compiled errorlist file.

DESCRIPTION

In some circumstances, it may be advantageous to customize the compilation error descriptions for translation into different languages or to suit personal tastes.

Two M2Sprint programs (M2E and M2Errors) use the special M2.errorlist file located in the M2Data: directory to obtain compiler error descriptions. This file cannot be easily modified, however its ASCII equivalent (M2Sprint_Errors) located in the "READ_ME drawer on "Library Sources 1" disk, can be.

The CompErrors.txt file is the ASCII equivalent of the M2.errorlist file. To make modifications, simply load the file into M2E or another text editor and change the descriptions as needed. The error numbers must be listed in numerically ascending order with no lines added or removed. Error descriptions must be no more than sixty characters and must be given on a single line.

The MakeErrorList utility is used to process the modified file and output a special errorlist file under a user specified name. M2Sprint will only recognize the new errorlist when it has been copied to the M2Data: directory under the M2.errorlist filename.

NOTE: If M2E has already loaded the error list file (a compilation has been done in the current session), the editor must be quit and restarted in order for the new error list to be recognized.

EXAMPLE

Error #50 can be changed from:

50 identifier not declared or not visible <-- standard message

to:

50 identifier not around

<-- modified message

NOTE: The errors must follow the format in the example. The error number must be listed first, followed by a single space, and then the description (not more than 60 characters long).

15. EasyBeeper

The purpose of the EasyBeeper module is to provide an easy means to control the Amiga's audio device in order to produce simple beeps and sounds of various tones and intensities often used to send warnings or draw the attention of the user.

One of the impressive features of the Amiga's audio device is its ability to be shared among several tasks in the system. Even if a full blown music composition program is running and using all four of the Amiga's audio channels, it is still possible to send a warning "beep" to the user. The secret lies in priorities.

Each request made to the audio device has a priority assigned to it. Higher priority requests are always processed before lower priority ones. The ROM Kernel manual suggests priorities for various applications: sound effects should have priorities in the range [-70..0], music programs should be in the range [-50..50], and "announciators" should be in the range [80..90]. The sounds produced by EasyBeeper are at priority 85.

The Amiga hardware provides four sound channels, two for the left and two for the right. The audio device can process sound requests on any of the four channels independently with each "sound request" indicating the preferred channel(s) for the output.

Various combinations may become impossible at some points because of other tasks in the system that are using the requested channels at a higher priority. To combat this problem, a list of preferred combinations can be submitted to the audio device which tries each combination until successful. If none work, no sound is produced.

Any of the following audio channel combinations is valid for Easy-Beeper:

Left #0 & Right #0

Left #1 & Right #1

Left #0

Right #0

Left #1

Right #1

Using EasyBeeper

In the simplest of cases, the Beep procedure can be imported from EasyBeeper and a call issued to it. The nextXXX variables may be modified to alter the kind of sound produced.

Example of usage

```
MODULE BeepTest;

FROM EasyBeeper IMPORT Beep, nextVolume;

BEGIN
    nextVolume:=32; (* half intensity *)
    Beep;            (* speaker beeps *)
END BeepTest.
```

Variables

NAME

nextVolume : CARDINAL;

DEFAULT VALUE

64. This is the maximum volume that a sound can have.

PURPOSE

This variable determines the volume at which the next sound will be produced. The value can range from 0 (no sound) up to 64 (maximum volume).

Procedures

NAME

Beep - Cause the Amiga to beep

SYNOPSIS

Beep

PURPOSE

Queues up a request to the audio device. The wave form used is two bytes long. Priority of the request is 85. Volume, duration and pitch are determined by the module's variables.

NAME

nextCycles : CARDINAL;

DEFAULT VALUE

90

PURPOSE

This variable indicates how many times the audio device should play the sound wave. The sound wave used by EasyBeeper is two bytes long. A value of 0 here means to play the sound forever (this should not be used). If you want to prolong the audio beep, make the nextCycles value larger.

16. EasyDBuf

The purpose of the EasyDBuf module is to provide an Intuition compatible method to perform very simple double-buffering. Double-buffering is a technique used to produce smooth animation.

The Amiga creates its displays by combining all adjacent bits from super-imposed bit-planes into a color register number. This color register number indicates a specific color register which contains the actual color to display on the screen for that pixel. This scheme is referred to as color indirection.

The image on the Amiga's monitor is created by an electron beam constantly being swept from left to right, top to bottom of the screen. This beam stimulates chemical products on the inside of the picture tube which in turn emit light. Depending on the chemical, the light emission varies in color so by aiming the video beam at the proper chemicals, a picture is created.

As mentioned above, the video beam sweeps the back of the picture tube from left to right, top to bottom. This process occurs 60 times per second on NTSC displays (in the US) and at 50 times per second on PAL displays (in Europe).

To produce animation, it is necessary to render display frames which provide transitions from one state to another. This is similar to drawing a bunch of pictures representing someone walking, and then flipping manually the pictures to simulate motion. On a computer this is done by presenting the user with a picture and quickly changing the displayed picture to something else.

This may seem simple in theory, but there is a problem. Visible video "glitches" can occur in the display if the video beam happens to be aimed at the same place where the computer is drawing. The double-buffering technique was developed to solve this very problem.

Using EasyDBuf

To perform double-buffering, you must:

1. Call CreateDBufScreen() to open the screen.

Before opening the double-buffered screen, you may wish to modify the nextXXX variables to alter the characteristics of the screen.

2. Do the rendering, flip to the other buffer, do more rendering, (repeat)

3. Call CloseDBufScreen() to close the screen and free the memory.

Example of usage

```
...
sp:=CreateDBufScreen(320,200,3); (* 320x200, 3 bit planes *)
IF sp # NIL THEN                (* Make sure screen opened *)
    rp:=ADR(sp^.RPort);          (* Get the screen rastport *)

    Delay(100);                   (* Wait 2 seconds *)
    Move(rp,100,100);             (* Move graphics pen *)
    Text(rp,ADR("Hello world!"),12);
    SwapDBufScreen(sp);          (* Display it *)
    Delay(200);                   (* Wait 4 seconds *)

    Move(rp,100,100);             (* Move graphics pen *)
    Text(rp,ADR("How are you?"),12);
    SwapDBufScreen(sp);          (* Display our msg *)
    Delay(200);                   (* Wait 4 seconds *)

    CloseDBufScreen(sp);          (* Close the screen *)
END;
...
```

The basic principle is simple: one image is displayed on screen while another is rendered in an area of memory off screen. Once the second image has been finished, the computer must be told to display it, and start the process of drawing the next frame in the same piece of memory the first image (now off screen) occupied. This process is repeated over and over to provide very smooth animation. One drawback however is that this double-buffering requires twice the amount of memory as simple animation due to the on-screen and off-screen memory.

The EasyDBuf module provides a simple means to create double-buffering. It uses an Intuition screen with two bitmaps (instead of the usual one bitmap). While rendering takes place in one bitmap, the other is displayed. Once the rendering is done, the buffers are swapped.

This module uses a special data structure which is kept in the UserData field of the screen structure. When using this module, you **must not** modify this field.

NAME

nextBlockPen : CARDINAL;

DEFAULT VALUE

1

PURPOSE

This variable defines the pen number used to render the title bar of the next screen opened.

Variables

NAME

nextDetailPen : CARDINAL;

DEFAULT VALUE

0

PURPOSE

This variable defines the pen number used to render the title bar of the next screen opened.

NAME

`nextFlags : ScreenFlagSet;`

DEFAULT VALUE

`ScreenFlagSet{ScreenQuiet}`. Open screen without a title bar.

PURPOSE

This variable defines several flags that will be used when opening the next screen. `ScreenBehind` can be specified to force the screen to open in the back of all screens. That screen can then be brought to the front using the `Intuition.ScreenToFront()` procedure.

NAME

`nextModes : ViewModeSet;`

DEFAULT VALUE

`ViewModeSet{}`. This means low-resolution display, non-interlace.

PURPOSE

This variable defines the modes of the next screen to be opened. Modes indicate how the hardware is to display the screen data. You can specify HiRes, Lace, GenLock, etc...

Procedures

NAME

CreateDBufScreen - Open a double-buffered screen

SYNOPSIS

CreateDBufScreen(width,height,depth:INTEGER): ScreenPtr;

width The screen's width in pixels.

height The screen's height in pixels.

depth The number of bit-planes the screen should have.

result A pointer to the screen, or NIL if it could not open.

PURPOSE

Creates a double-buffered screen. The screen will have the characteristics described by the "nextXXX" variables. The return value from this procedure points to the new screen structure, or will be NIL if the screen could not be opened (probably due to lack of memory).

If you wish your software to support various display formats (NTSC or PAL), the height parameter should be set to Intuition.StdScreenHeight (-1).

Current Amiga hardware supports from 1 to 6 bit-planes depending on the resolution.

The double-buffered screen is closed by calling the CloseDBufScreen() procedure.

NAME

nextTextAttr : TextAttrPtr;

DEFAULT VALUE

NIL

PURPOSE

This variable defines the default font to use when opening the next screen. If this parameter is NIL, then the current default system font will be used. If a value is specified for this variable, the screen title, the window titles and the menu titles will all be rendered in this font. Note that the font must already be in memory when opening the screen; i.e. Intuition will not load fonts from disk.

NAME

SwapDBufScreen - Swap the display buffers of a double-buffered screen (show the hidden view).

SYNOPSIS

```
SwapDBufScreen(screen:ScreenPtr);
```

screen Double-buffered screen for which to swap buffers.

PURPOSE

Display the off-screen buffer. This is where the double-buffering actually happens. This causes the off-screen bitmap to become visible, and the on-screen bitmap to become invisible. It also changes some pointers to rastports so that your rendering will end up in the correct buffer.

NAME

CloseDBufScreen - Close a double-buffered screen

SYNOPSIS

```
CloseDBufScreen(screen:ScreenPtr);
```

screen The double-buffered screen to close. Typically this will be currentDBufScreen.

PURPOSE

Free the memory used by a double-buffered screen. The screen parameter will have been obtained by a previous call to CreateDBufScreen().

17. EasyGadgets

The purpose of the EasyGadgets module is to provide procedures to aid in the creation of attractive and functional displays using the standard Amiga gadgets.

Intuition provides gadgets to serve as one of the main centers of interaction between computer and user. Gadgets come in several flavors: boolean, string, integer and proportional. Every type of gadget has its own data structure which needs to be correctly initialized before the gadget can be used. Initializing all these structures is tedious, and can easily lead to bugs in programs. It is also difficult to modify and maintain gadget initializing code.

The EasyGadgets module provides procedures to create all types of gadgets, and even allows you to add borders and drop shadows to the gadgets.

NAME

SetDBufScreenColor - Adjust the display colors of a double-buffered screen.

SYNOPSIS

```
SetDBufScreenColor(screen:ScreenPtr; colorNumber:CARDINAL;  
                    red,green,blue:CARDINAL);
```

screen	The double-buffered screen to set the colors for.
colorNumber	The pen number to change the color of.
red,green,blue	New color values for the given colorNumber.

PURPOSE

Allows you to easily change the color values for any given pen number. The red, green and blue parameters are what composes an Amiga color. These values can range from 0 (no color) to 15 (full color).

```
CloseWindow(wp);  
DisposeList(currentList);  
END;  
..
```

```
(* Close the window *)  
(* Free mem used by gadgets*)
```

Using EasyGadgets

To create gadgets:

1. Create a window that will hold the gadgets.
2. Call `StartList()` to initialize the module's variables.
3. Do any combination of procedure calls from this module in order to generate your gadget list.
4. Install the gadgets in your window using `AddGList()`.
5. Cause Intuition to render your gadgets in the window using `RefreshGList()`.
6. Close the window.
7. Call `DisposeList()` to free the memory used by the gadget list.

Example of usage

...

```
wp:=CreateWindow(10,10,350,75,"An Easy Window!", IDCMPFlagSet{},
WindowFlagSet{},NIL,NIL) THEN
IF wp # NIL THEN

    StartList();                (* Start a new list      *)
    nextIntuiFrontPen:=1;       (* Set up some defaults *)
    nextBorderFrontPen:=3;

    AddBoolGadget(20,20,50,13,"Hello"); (* Add a boolean gadget *)
    AddBorders();              (* Give it borders      *)
    AddDropShadow();           (* Give it a shadow      *)

    AddStrGadget(20,50,75,8,35,"Yo!"); (* Add a string gadget  *)
    AddBorders();              (* Give it a border      *)

    AddPropGadget(125,20,88,40,TRUE,TRUE,32767,32767,8192,9216);
    AddBorders();

    IF NOT listFailed THEN      (* Gadgets allocated?  *)
        dummy:=AddGList(wp,currentList,0,-1,NIL);
        RefreshGList(currentList,wp,NIL,-1);
        Delay(750);
    END;
```

NAME

`currentList : GadgetPtr;`

DEFAULT VALUE

None

PURPOSE

After a gadget list is allocated, this variable has a pointer to the first gadget in the list. This pointer can be passed to `Intuition.AddGList()` to install the gadgets in a window.

Variables

NAME

listFailed : BOOLEAN;

DEFAULT VALUE

None

PURPOSE

After having allocated a gadget list, this variable will tell you whether or not everything you asked for was added. If the value is FALSE, the list is not valid and should not be used. The only reason why a gadget list would not be correctly allocated is if the system ran out of memory.

NAME

currentStrBuffer : STRPTR;

DEFAULT VALUE

None

PURPOSE

This variable points to the last buffer allocated for a string gadget.

NAME

currentGadget : GadgetPtr;

DEFAULT VALUE

None

PURPOSE

This variable has a pointer to the last gadget added by one of the procedures in this module. This may be useful when you want to modify the record to use some feature that this module does not support directly.

Note that even if the last gadget allocation failed, this variable will always point to a valid gadget record. This allows you to allocate a complete gadget list, modifying gadgets and intuit text as you go along, and only when you finish, check the listFailed variable once to confirm that everything was allocated.

NAME

`currentIntui : IntuiTextPtr;`

DEFAULT VALUE

None

PURPOSE

This variable has a pointer to the last `IntuiText` created by `AddBoolGadget()`. This may be useful when you want to modify the record to use some feature that this module does not support directly.

Note that even if the last allocation failed, this variable will always point to a valid `IntuiText` record. This allows you to allocate a complete gadget list, modifying gadgets and intui text as you go along, and only when you finish, check the `listFailed` variable once to confirm that everything was allocated.

NAME

currentIntBuffer : STRPTR;

DEFAULT VALUE

None

PURPOSE

This variable points to the last buffer allocated for an integer gadget.

NAME

nextIntActivation : ActivationFlagSet;

DEFAULT VALUE

ActivationFlagSet{RelVerify,StringCenter}

PURPOSE

Determines the Activation flags to use for the next integer gadget created. See the Intuition manual to learn about Activation flags.

NAME

nextBoolActivation : ActivationFlagSet;

DEFAULT VALUE

ActivationFlagSet{RelVerify}

PURPOSE

Determines the Activation flags to use for the next boolean gadget created. See the Intuition manual to learn about Activation flags.

NAME

nextStrActivation : ActivationFlagSet;

DEFAULT VALUE

ActivationFlagSet{RelVerify}

PURPOSE

Determines the Activation flags to use for the next string gadget created. See the Intuition manual to learn about Activation flags.

NAME

nextPropActivation : ActivationFlagSet;

DEFAULT VALUE

ActivationFlagSet{RelVerify}

PURPOSE

Determines the Activation flags to use for the next proportional gadget created. See the Intuition manual to learn about Activation flags.

NAME

nextFlags : GadgetFlagSet;

DEFAULT VALUE

GadgHComp

PURPOSE

Determines the Gadget flags to use for the next gadget created. See the Intuition manual to learn about Gadget flags.

NAME

nextIsReqGadget : BOOLEAN;

DEFAULT VALUE

FALSE

PURPOSE

Specifies if the next gadget created will be a requester gadget or a normal gadget. This should be TRUE to indicate that it is a requester gadget, and FALSE otherwise.

NAME

nextIntuiTextAttr : TextAttrPtr;

DEFAULT VALUE

NIL

PURPOSE

Determines the text attributes (font, style, size) of the next Intui-Text created to go with a boolean gadget. NIL means that the current default font should be used.

NAME

nextID : CARDINAL;

DEFAULT VALUE

Set to 0 every StartList()

PURPOSE

Determines the value of GadgetID for the next gadget created. This value is automatically incremented by one every time you add a gadget.

NAME

nextIntuiBackPen : INTEGER;

DEFAULT VALUE

1

PURPOSE

Determines the background color used for the next IntuiText created to go with a boolean gadget.

NAME

nextIntuiFrontPen : INTEGER;

DEFAULT VALUE

2

PURPOSE

Determines the foreground color used for the next IntuiText created to go with a boolean gadget.

NAME

nextIntuiLeftEdge : INTEGER;

DEFAULT VALUE

5

PURPOSE

Determines the offset from the left edge of the next boolean gadget created at which the associaed IntuiText should appear.

NAME

nextIntuiDrawMode : DrawingModeSet;

DEFAULT VALUE

Jam1

PURPOSE

Determines the drawing mode used for the next IntuiText created to go with a boolean gadget.

NAME

nextBorderFrontPen : INTEGER;

DEFAULT VALUE

3

PURPOSE

Determines the foreground color used for the next Border created.

NAME

AddIntGadget - Add an integer gadget to the current gadget list

SYNOPSIS

```
AddIntGadget(leftEdge,topEdge,width,height:INTEGER;  
              bufSize:CARDINAL; initialValue:LONGINT);
```

leftEdge,topEdge	Position of gadget relative to window or requester.
width,height	Dimensions of the select box of the gadget.
bufSize	Number of digits that the Int gadget should allow.
initialValue	The initial value that the gadget will display when first rendered.

PURPOSE

Integer gadgets are exactly the same as normal string gadget, except they only allow the user to type in digits, no alphabetic character is allowed.

You must specify the coordinates for the select box of your gadget. The coordinates are relative to the upper left corner of the display element in which the gadget is (window, requester).

You must specify the maximum number of digits allowed in the integer gadget. If you limit the number of digits to 2, for example, the user will not be able to enter a number larger than 99 in the integer gadget.

You must also specify the initial (default) value of the integer gadget. This is what the gadget will initially contain the first time it is displayed.

NAME

nextShadowFrontPen : INTEGER;

DEFAULT VALUE

2

PURPOSE

Determines the foreground color used for the next shadow created.

NAME

DisposeList - Deallocate all memory allocated for a particular gadget list

SYNOPSIS

DisposeList(gadgetList:GadgetPtr);

gadgetList Gadget list to deallocate, allocated by calls in this module.

PURPOSE

Every time your program creates a gadget, this module allocates a chunk of memory to hold the record that is needed. Once you have finished with a gadget list, it is necessary to return the memory used to the system.

The value you pass this procedure will typically be that of **currentList**.

NAME

AddBoollImageGadget - Add a boolean gadget that uses an image for rendering to the current gadget list.

SYNOPSIS

```
AddBoollImageGadget(leftEdge,topEdge,width,height:INTEGER;  
                      image:ImagePtr);
```

leftEdge,topEdge	Position of gadget relative to window or requester.
width,height	Dimensions of the select box of the gadget.
image	Pointer to an Image record used to render the gadget.

PURPOSE

Boolean gadgets are the most commonly used type of gadgets. They are used for things like OK/Cancel gadgets. All system gadgets on a window (close gadget, sizing gadget, etc..) are boolean gadgets.

You must specify the coordinates for the select box of your gadget. The coordinates are relative to the upper left corner of the display element in which the gadget is (window, requester).

This procedure creates a special variety of boolean gadgets. Instead of being identified by text titles, these gadgets are identified by graphical images. The system gadgets of a window are rendered this way. You must supply this routine a properly initialized Image structure. This record may have been created by the IFF2Obj utility included with M2Sprint, or may have been created by another procedure in your application.

NAME

AddPropGadget - Add a proportional gadget to the current gadget list

SYNOPSIS

```
AddPropGadget( leftEdge,topEdge,width,height:INTEGER;  
                moveHoriz,moveVert:BOOLEAN;  
                hPot,vPot,hBody,vBody:CARDINAL);
```

leftEdge,topEdge	Position of gadget relative to window or requester.
width,height	Dimensions of the select box of the gadget.
moveHoriz	Gadget knob can move horizontally.
moveVert	Gadget knob can move vertically.
hPot,vPot	Horizontal and Vertical potentials initially represented by the gadget.
hBody,vBody	Horizontal and Vertical amount of data visible.

PURPOSE

Proportional gadgets are used to implement scroll bars on windows. An example of such are the Workbench windows.

You must specify the coordinates for the select box of your gadget. The coordinates are relative to the upper left corner of the display element in which the gadget is (window, requester).

You must specify if this gadget can move horizontally or vertically, or both. For a scroll bar on the right side of a window, the scroll bar can be moved vertically, but not horizontally. For the scroll bar on the bottom of windows, it can move horizontally, but not vertically. For the gadget in Preferences which allows you to center your display on your monitor, it can move in both directions.

Consult the Intuition manual to learn about the remaining 4 variables. They control the size and position of the knob of the gadget within the gadget select box.

NAME

AddStrGadget - Add a string gadget to the current gadget list

SYNOPSIS

```
AddStrGadget(leftEdge,topEdge,width,height:INTEGER;  
              bufSize:CARDINAL;  
              initialString:ARRAY OF CHAR);
```

leftEdge,topEdge	Position of gadget relative to window or requester.
width,height	Dimensions of the select box of the gadget.
bufSize	Number of characters that the string gadget should allow.
initialString	The initial string that the gadget will display when first rendered, may be "".

PURPOSE

The Workbench Rename window is an example of a string gadget. It is an area of the display which can contain text and a text cursor and which allows the user to modify this text. This is a very useful tool when you want some input from the user.

You must specify the coordinates for the select box of your gadget. The coordinates are relative to the upper left corner of the display element in which the gadget is (window, requester).

You must specify the maximum number of characters allowed in the string gadget. This limits the length of strings that can be entered in the string gadget by the user.

You must also specify the initial (default) value of the string gadget. This is what the gadget will initially contain the first time it is displayed.

NAME

AddBorders - Add a border around the last gadget added

SYNOPSIS

AddBorders();

PURPOSE

This procedure adds a border around the last gadget that was created. This is often desirable to properly define your gadget's select box.

NAME

AddDropShadow - Add a drop shadow to the bottom right of the last gadget added

SYNOPSIS

AddDropShadow();

PURPOSE

This routine will add a shadow in the right bottom corner of the last gadget that was created. This gives the impression that the gadget is slightly raised above the screen.

This routine takes into account whether or not the gadget has already a border around it.

18. EasyGamePort

The purpose of the EasyGamePort module is to provide procedures to read data originating from joysticks plugged into the Amiga's game ports.

Current Amiga hardware offers two general-purpose game ports. One of these is generally used for the mouse, and the other often receives a joystick. The Amiga OS provides the gameport device to read joystick data. Unfortunately, using the game port device requires an understanding of how device IO functions and in particular, how the game port device functions. EasyGamePort greatly eases the task of reading events from either Amiga game ports.

Using EasyGamePort

To read the joystick values you must:

1. Declare three variables (2 INTEGER and 1 BOOLEAN).
2. Call ReadJoy() or WaitJoy() passing as parameters the variables.
3. Interpret the new values of the variables.

Example of usage

...

LOOP

 ReadJoy(l,x,y,button); (* Read port #1 *)

 WriteInt(x,5); WriteInt(y,5); (* Returned values *)

 IF button THEN

 WriteString("-- Button --"); (* Button depressed *)

 ELSE

 WriteString("-- No Button --"); (* Not depressed *)

 END;

 WriteLn;

END;

...

Variables

NAME

joyFailed : BOOLEAN;

DEFAULT VALUE

None

PURPOSE

This variable holds state information relating to the last operation performed on the game port device. If the last operation failed, this value will be FALSE, otherwise it will be TRUE.

Possible cause of errors include lack of memory, and inability to obtain signal bits.

Procedures

NAME

ReadJoy - Read the current state of a joystick.

SYNOPSIS

```
ReadJoy(port:CARDINAL; VAR x,y:INTEGER;  
        VAR button:BOOLEAN);
```

- | | |
|--------|---|
| port | The port from which to read. This can be either 0 (the mouse port) or 1. |
| x | This variable receives a value describing the current horizontal orientation of the joystick. (-1) means the stick is pushed to the left, (0) means the stick is centered and (1) means the stick is pushed to the right. |
| y | This variable receives a value describing the current vertical orientation of the joystick. (-1) means the stick is pushed to the top, (0) means the stick is centered and (1) means the stick is pushed to the bottom. |
| button | This variable receives a value describing the current state of the joystick button. A value of TRUE means the button is down, FALSE means it is up. |

PURPOSE

This procedure is used to read a game port and return immediately. This will be used when your program has some tasks to perform, but also wants to monitor what the user is doing.

For example, in a game, the program needs to detect the direction of the joystick, and then continue running the program taking into account the joystick direction.

The parameters 'x', 'y' and 'button' are used as outputs. 'x' and 'y' will receive values in the range [-1..1]. 'button' will be TRUE if the joystick button is depressed, or FALSE if it is not.

NAME

WaitJoy - Wait for the user to move the joystick.

SYNOPSIS

```
WaitJoy(port:CARDINAL; VAR x,y:INTEGER;  
        VAR button:BOOLEAN);
```

- port** The port from which to read. This can be either 0 (the mouse port) or 1.
- x** This variable receives a value describing the current horizontal orientation of the joystick. (-1) means the stick is pushed to the left, (0) means the stick is centered and (1) means the stick is pushed to the right.
- y** This variable receives a value describing the current vertical orientation of the joystick. (-1) means the stick is pushed to the top, (0) means the stick is centered and (1) means the stick is pushed to the bottom.
- button** This variable receives a value describing the current state of the joystick button. A value of TRUE means the button is down, FALSE means it is up.

PURPOSE

This procedure is used to wait for the user to perform some action with the joystick. This will be used when your program requires input from the user and can't progress without it. For example, to start the game (press 'Fire' to start) or to enter the user's initials in the high score table of a game.

The parameters 'x', 'y' and 'button' are used as outputs. 'x' and 'y' will receive values in the range [-1..1]. 'button' will be TRUE if the joystick button is depressed, or FALSE if it is not.

19. EasyGels

The purpose of the EasyGels module is to provide procedures to create and manipulate Amiga GELS (Graphics ELementS) used in animation. These include BOBS (Blitter OBjectS) and VSprites (Virtual Sprites).

The Amiga's custom hardware makes it uniquely qualified in the field of personal computer animation. Two important hardware subsystems in this area are the blitter and the sprite generator.

Blitter stands for "Bit-Block Transfer". The blitter's job in the Amiga system is to move large chunks of data from one memory location to another. It is optimized for this task and can thus perform it faster than the normal Amiga CPU. The blitter also understands the concept of rectangles and can clip these on arbitrary bit boundaries in memory. While moving data, the blitter can apply logic operations on it, such as complement. The blitter can also draw lines at any angle with great speed.

A sprite is a bitmap image which is independently controlled from the background image. A sprite can be overlaid on top of the background without affecting said background. You can move the sprite without worrying about what's in back of it. Sprites can perform collision detection to learn when two sprites overlap. This is very useful when designing games. An example of a sprite is the Amiga mouse pointer.

The blitter is used throughout the Amiga OS to quickly render lines (window borders for example), and also to move large amounts of screen data (when moving windows for example). The blitter is the reason why the Amiga's user interface (menus, windows) is so responsive when compared with other machines.

With some work, it is possible to use the blitter to create animation. By copying data from off-screen buffers to on-screen buffers you can simulate motion.

The Amiga's operating system provides several routines to use the blitter as an animation engine. The OS manages virtual objects known as Blitter Objects (BOBS). You can use BOBS in virtually any Amiga display. Dragging icons on the Workbench is an example of simple animation with BOBs.

Unfortunately, creating a BOB requires many records to be initialized and maintained. This can make incorporating BOBS in your programs complicated and often tedious. The routines in this module will greatly ease the use of BOBS in your applications.

Unlike sprites, BOBs are very dependant on the background. BOBs are rendered using the background display's memory. When you render a BOB on the screen, it's image is blitted from an off-screen buffer to the on-screen buffer. This destroys the background image. BOBs provide a special feature which will preserve the background behind a BOB. When you move the BOB, newly uncovered regions of the background get restored, and newly hidden regions get saved. For this to be done, all you need to do is to provide an additional off-screen buffer area where the background can be preserved. This buffer should be the same size as the BOB itself.

This module also provides support for VSprites. Procedures very similar to the BOB procedures are available.

Consult the Amiga ROM Kernel manual to learn about the format of the data required by BOBs and VSprites.

Using EasyGels

To use EasyGels, you must:

1. Create a display element such as a screen.
2. Attach a GelsInfo record to the display's rastport using `CreateGelsInfo()`.
3. Allocate some ram to hold the BOB's image using `CreateBuffer()`.
4. If the BOB has to preserve the background, allocate a separate buffer to hold the background when the bob is on screen. Use `CreateBuffer()` and supply the same parameters as for the BOB's data buffer.
5. Create the BOB itself using `CreateBob()`.
6. Add the BOB to the rastport's gels list using `Gels.AddBob()`.
7. Cause the system to render the BOB using `Gels.DrawGList()`.
8. Remove the BOB from the display using `Gels.RemlBob()`.
9. Dispose of the BOB using `DisposeBob()`.
10. Deallocate the memory used by the BOB's data and off-screen buffers using `DisposeBuffer()`.

Example of usage

```
...
sp:=CreateScreen(ScrWidth,ScrHeight,ScrDepth,"An Easy Screen!");
IF sp # NIL THEN

    rp:=ADR(sp^.RPort);          (* Get rastport      *)
    IF CreateGelsInfo(rp) THEN   (* Attach a gels info *)

        data:=CreateBuffer(BobWidth,BobHeight,BobDepth);
        IF data # NIL THEN      (* bob data          *)

            save:=CreateBuffer(BobWidth,BobHeight,ScrDepth);
            IF save # NIL THEN   (* background buffer *)

                bob:=CreateBob(100,100,BobWidth,BobHeight,
                               BobDepth,data,save,NIL,BITSET{0,1},BITSET{});
                IF bob # NIL THEN

                    AddBob(bob,rp);          (* put BOB on screen *)
                    DrawGList(rp,ADR(sp^.VPort));
                    Delay(100);

                    bob^.BobVSprite^.X:=200; (* Change position  *)
                    DrawGList(rp,ADR(sp^.VPort));
                    Delay(100);

                    RemIBob(bob,rp,ADR(sp^.VPort));
                    DisposeBob(bob);         (* Delete BOB       *)
                END;

                DisposeBuffer(save);         (* Return background mem*)
            END;
            DisposeBuffer(data);             (* Return bob data    *)
        END;

        DisposeGelsInfo(rp);               (* Remove gels info   *)
    END;

    CloseScreen(sp);                      (* Close screen       *)
END;
...
```

Variables

NAME

nextVFlags : VSpriteFlagSet;

DEFAULT VALUE

VSpriteFlagSet{Overlay}

PURPOSE

This variable defines various attributes that the next VSprite created will have. Consult the ROM Kernel manual to learn what these flags are.

NAME

nextBFlags : BobFlagSet;

DEFAULT VALUE

BobFlagSet{ }

PURPOSE

This variable defines various attributes that the next BOB created will have. Consult the ROM Kernel manual to learn what these flags are.

Procedures

NAME

CreateBob - Create a BOB

SYNOPSIS

```
CreateBob(x,y:INTEGER; width,height,depth:CARDINAL;  
          imageData,saveBuffer,dbufBuffer:ADDRESS;  
          planePick,planeOnOff:BITSET): BobPtr;
```

x,y	Position of the bob in the RastPort (in pixels).
width,height	Dimensions of the bob (in pixels).
imageData	Address of a buffer containing the data for the bob. This buffer must be in CHIP memory. You can use CreateBuffer() to obtain the memory.
saveBuffer	Optional buffer used by the system to preserve the background behind your bob. Once the bob is moved, the background is restored. This buffer must be of the same size as the imageData and must have the same number of planes as the destination RastPort.
dbufBuffer	Optional buffer used by the system when your bob is double-buffered. The same requirements as saveBuffer apply.
planePick,planeOnOff	Refer to the comments in Intuition.def (M2Sprint Library Module Reference) on the Image record to learn about how these two variables (also found in the Image record) influence the rendering.
result	A pointer to a newly created bob, or NIL if not enough memory.

PURPOSE

This routine takes the input parameters and does all the necessary work to create a BOB. Normally, a pointer to a BOB record is returned. If there was not enough memory, NIL will be returned.

NAME

DisposeBob - Free the memory used by a BOB

SYNOPSIS

DisposeBob(bob:BobPtr);

bob The bob to delete, allocated by CreateBob().

PURPOSE

Returns allocated memory to the system. When you create a BOB using CreateBob(), some memory is allocated from the system to store the BOB data. It is very important that you return all memory allocated to the system before exiting your programs.

NAME

CreateGelsInfo - Add a GelsInfo to a rastport.

SYNOPSIS

CreateGelsInfo(rp:RastPortPtr): BOOLEAN;

rp RastPort onto which to attach a GelsInfo.

result TRUE on success, or FALSE if failure.

PURPOSE

A GelsInfo is a record used by the system to manipulate Gels. Before you can add BOBS or VSprite to a rastport, this rastport must have its own GelsInfo record.

The return value will be TRUE if all is OK, or FALSE if there was not enough memory to allocate a GelsInfo record.

NAME

DisposeGelsInfo - Deallocate the memory used by a GelsInfo

SYNOPSIS

DisposeGelsInfo(rp:RastPortPtr);

rp The RastPort onto which the GelsInfo is attached
 (by CreateGelsInfo).

PURPOSE

Returns allocated memory to the system. When you create a GelsInfo record using CreateGelsInfo(), some memory is allocated from the system to store the GelsInfo data. It is very important that you return all memory allocated to the system before exiting your programs.

NAME

CreateVSprite - Create a new VSprite

SYNOPSIS

CreateVSprite(x,y:INTEGER; height:CARDINAL;
image:ADDRESS; colors:ADDRESS): VSpritePtr;

x,y	Position of the VSprite within the RastPort (in pixels).
height	Height of the VSprite in pixels. VSprite are a fixed width (16 pixels).
image	A CHIP memory buffer containing the VSprite rendering data.
color	The color table describing the colors desired for this VSprite.
result	A pointer to a newly created VSprite, or NIL if not enough memory.

PURPOSE

This routine allocates a VSprite record and initializes it to the values passed as parameters. You can then add this VSprite to a RastPort gels list and display it.

NAME

DisposeVSprite - Deallocate the memory used by a VSprite

SYNOPSIS

DisposeVSprite(vsprite:VSpritePtr);

vsprite The vsprite to delete, allocated by CreateVSprite().

PURPOSE

Returns allocated memory to the system. When you create a VSprite record using CreateVSprite(), some memory is allocated from the system to store the VSprite record. It is very important that you return all memory allocated to the system before exiting your programs.

NAME

CreateBuffer - Create a plane buffer

SYNOPSIS

CreateBuffer(width,height,depth:CARDINAL): ADDRESS;

width,height	Dimensions of plane buffer.
depth	Number of bit-planes in the plane buffer.
result	The address of a plane buffer, or NIL if not enough memory.

PURPOSE

You can use CreateBuffer() to allocate the buffers needed for BOBS or VSprites. Simply give this routines the same dimension parameters as to CreateBob() or CreateVSprite(), and the necessary memory will be allocated.

NAME

DisposeBuffer - Deallocate the memory used by a plane buffer

SYNOPSIS

DisposeBuffer(buffer:ADDRESS);

buffer The plane buffer allocated by CreateBuffer().

PURPOSE

Returns allocated memory to the system. When you create a plane buffer using CreateBuffer(), some memory is allocated from the system to hold the plane buffer. It is very important that you return all memory allocated to the system before exiting your programs.

20. EasyIDCMP

The purpose of the EasyIDCMP module is to provide a single procedure that greatly simplifies the handling of Intuition messages sent via a window's IDCMP.

The Amiga is a multitasking computer which provides great flexibility to the user. Unfortunately, this flexibility comes with a cost to the programmer, complexity. The Amiga is a much more difficult machine to program than plain vanilla single-tasking systems. Thus, the learning curve when first starting on the system can be very steep.

One of the most important aspects of a multitasking kernel is the need for inter-task communications. On the Amiga, several mechanisms are provided to aid in the area, some of these being signals, message ports and semaphores.

Intuition is the component of the Amiga OS which handles user-interface issues. It manages the windows, menus, gadgets, and mouse part of the system. Intuition runs asynchronously with user tasks. That is to say, that it runs 'at the same time' or 'in parallel' with other system tasks.

Since Intuition has the job of informing programs of user activities, it must use the inter-task communication mechanisms provided by Exec. Every Intuition window has an Exec message port attached to it. This message port is used by Intuition to send messages to the "owner" of the window (a program) detailing, among other things, information about actions performed by the user. This special message port is known as IDCMP (Intuition Direct Communication Message Port).

When a window is opened, your program must specify which of the special IDCMP events it needs to know about. Some examples of IDCMP events that your program may need to know about are the user clicking the close gadget of your program's window, the user removing a disk from a disk drive, or activating/deactivating a window.

Often, the most complex part of a program is the main loop that handles user events. This loop is often hard to understand, modify or maintain and so requires great amounts of thought before its creation.

The procedure in the EasyIDCMP module greatly eases the job of designing your programs to handle IDCMP events. It dispatches program control to specific procedures provided to handle every possible event that arrive at an IDCMP port. The ProcessEvents() procedure handles the details involved with getting IDCMP events and sending them back to Intuition. All you need to do is provide a set of procedures containing the instructions to be executed when an event occurs that will be automatically called when an IDCMP event happens.

Using EasyIDCMP

To use EasyIDCMP, you must:

1. Determine which events are expected in a window.
2. Write procedures to handle every one of these events.
3. Initialize a ProcTable record with pointers to your procedure.
4. Open a window.
5. Call ProcessEvents(). At this point, the set of procedures you have written will get called automatically whenever an Intuition event occurs.
6. Once ProcessEvents() returns, you can close your window.

Example of usage

```
...
PROCEDURE HandleCloseWindow (VAR intuit: IntuiMessage): INTEGER;
BEGIN
    (* This gets called whenever the close gadget is hit *)
END HandleCloseWindow;
...
WITH procTable DO
    WaitForEvent := TRUE;           (* Wait for events      *)
    MenuPick     := HandleMenuPick;
    CloseWindow  := HandleCloseWindow; (* Our procedure    *)
    GadgetUp     := HandleGadgetUp;
    GadgetDown   := HandleGadgetDown;
END;

wp:=CreateWindow(180,40,280,120,"A Window!",
    IDCMPFlagSet{CloseWindowFlag,GadgetUp,
        GadgetDown,MenuPick},
    WindowFlagSet{WindowClose,WindowDrag,
        WindowDepth,Activate, NoCareRefresh}
        +SmartRefresh,NIL,NIL);

IF wp # NIL THEN
    SetMenuStrip(wp,currentStrip);

    result:=ProcessEvents(wp,procTable);  (* handle the events *)

    ClearMenuStrip(wp);
    CloseWindow(wp);
END;
...
```


Procedures

NAME

ProcessEvents - Process the IDCMP events for a window

SYNOPSIS

ProcessEvents(wp:WindowPtr; VAR table:ProcTable): INTEGER;

wp Window for which to process the events.

table An initialized procedure table.

result 0 if there were no messages to read (if WaitForEvent is FALSE) otherwise, returns the error code generated by calling the procedures from the procedure table.

PURPOSE

This is the main IDCMP event dispatcher. In order for ProcessEvents to take care of your program's events, it requires the WindowPtr for the window to act on.

This routine also needs an initialized procedure table. Whenever an IDCMP event occurs, ProcessEvents dispatches the event handling to the appropriate procedure indicated in the procedure table.

The field "WaitForEvent" in the process table record indicates whether this routine should wait to receive an IDCMP event, or simply check if there is one available and return immediately if not. This may be useful if your program has to do some work when there are no Intuition events. Typically, WaitForEvent will be TRUE.

When procedures from the procedure table are called, they are passed the IntuiMessage record obtained from Intuition. This message contains all the information pertaining to the event that just occurred. The procedure you provide should return 0 to indicate success. Any other value is interpreted as an error message. When ProcessEvents() receives a non-zero return, it terminates by returning the value to the caller. This scheme is useful for simple error propagation, or for terminating the program on defined events (for example, the procedure that handles CloseWindow events can return 1 to indicate that the program is to terminate).

21. EasyMenus

The purpose of the EasyMenus module is to provide several procedures to easily create attractive and functional menu layouts using the standard Amiga menu facility.

Intuition provides very flexible menu handling. In order to allow such flexibility, the menu subsystem relies on various records to define the appearance and behavior of the menu strips. Initializing all the necessary records is tedious and produces hard to read code. Generally, the initialization code is also hard to modify because to make one modification often requires several more records to also be adjusted. Additionally, if you make an error initializing the records, the system will crash.

EasyMenus provides simple procedures that allow you to create any type of menu display in an easy to write and maintain style.

Using EasyMenus

To use EasyMenus you must:

1. Create a window that will hold the menu strip.
2. Call StartStrip() to initialize all the module's variables.
3. Do the required AddMenu(), AddItem() and AddSub() calls in order to create your menu strip.
4. Install the menu strip in the window using Intuition.SetMenuStrip().
5. Remove the menu strip from the window using Intuition.ClearMenuStrip().
6. Dispose of the memory used by the menu strip by calling DisposeStrip().
7. Close the window.

Example of usage

...

```
wp:=CreateWindow(10,10,350,75,"An Easy Window!",IDCMPFlagSet{},
                WindowFlagSet{},NIL,NIL);
IF wp # NIL THEN

    StartStrip();                (* Start a new menu strip      *)

    AddMenu("Project",100);
        AddItem("New","N");
        AddItem("Open",OC);
            AddSub("Document","D");
            AddSub("Graphics","G");
        AddItem("Save","S");

    IF NOT stripFailed THEN      (* Were all records allocated? *)
        SetMenuStrip(wp,currentStrip);
        Delay(1000);
        ClearMenuStrip(wp);
    END;
```

```
    DisposeStrip(currentStrip); (* Free memory of menu strip *)
    CloseWindow(wp);            (* Close the window *)
END;
...
```

Variables

NAME

stripFailed : BOOLEAN;

DEFAULT VALUE

None

PURPOSE

After having allocated a menu strip, this variable indicates whether or not everything requested was added. If the variable is FALSE, the strip is not valid and therefore should not be used. The only reason why a menu strip would not be correctly allocated is if the system ran out of memory.

NAME

`currentStrip : MenuPtr;`

DEFAULT VALUE

None

PURPOSE

After a menu strip has been allocated, this variable has a pointer to the first menu in the strip. This pointer can be passed to `Intuition.SetMenuStrip()` to install the menu strip in a window.

NAME

currentMenu : MenuPtr;

DEFAULT VALUE

None

PURPOSE

This variable has a pointer to the last menu added by the AddMenu() procedure. This may be useful when the record needs to be modified to use some feature that the EasyMenus module does not support directly.

Note that even if the last menu allocation failed, this variable will always point to a valid menu record. This allows you to allocate a complete menu strip, modifying menu, items and subitems as you go along, and only when you finish, check the stripFailed variable once to confirm that everything was allocated.

NAME

`currentItem : MenuItemPtr;`

DEFAULT VALUE

None

PURPOSE

This variable has a pointer to the last menu item added by the `AddItem()` procedure. This may be useful when the record needs to be modified to use some feature that the EasyMenus module does not support directly.

Note that even if the last item allocation failed, this variable will always point to a valid item record. This allows you to allocate a complete menu strip, modifying menu, items and subitems as you go along, and only when you finish, check the `stripFailed` variable once to confirm that everything was allocated.

NAME

currentSub : MenuItemPtr;

DEFAULT VALUE

None

PURPOSE

This variable has a pointer to the last menu subitem added by the AddSub() procedure. This may be useful when you want to modify the record to use some feature that this module does not support directly.

Note that even if the last subitem allocation failed, this variable will always point to a valid subitem record. This allows you to allocate a complete menu strip, modifying menu, items and subitems as you go along, and only when you finish, check the stripFailed variable once to confirm that everything was allocated.

NAME

`currentIntui : IntuiTextPtr;`

DEFAULT VALUE

None

PURPOSE

This variable has a pointer to the last `IntuiText` created by either `AddSub()` or `AddItem()`. This may be useful when you want to modify the record to use some feature that this module does not support directly.

Note that even if the last allocation failed, this variable will always point to a valid `IntuiText` record. This allows you to allocate a complete menu strip, modifying menu, items and subitems as you go along, and only when you finish, check the `stripFailed` variable once to confirm that everything was allocated.

NAME

nextMenuLeftEdge : INTEGER;

DEFAULT VALUE

5

PURPOSE

Determines the offset from the left side of the display for the next menu created. This value is automatically incremented every time you add a new menu. You can change this value to change the positioning of your menus.

NAME

nextItemLeftEdge : INTEGER;

DEFAULT VALUE

-1

PURPOSE

Determines the offset from the left edge of the current menu for the next item created.

NAME

nextSubLeftEdge : INTEGER;

DEFAULT VALUE

50

PURPOSE

Determines the offset from the left edge of the current item for the next subitem created.

NAME

nextIntuiLeftEdge : INTEGER;

DEFAULT VALUE

1

PURPOSE

Determines the offset from the left edge of the next item or subitem for the IntuiText associated with it.

NAME

nextItemFlags : ItemFlagSet;

DEFAULT VALUE

ItemFlagSet{ItemText,ItemEnabled}

PURPOSE

Determines the flags used for the next item created. See the Intuition manual for a definition of all the possible item flags.

NAME

nextSubTopEdge : INTEGER;

DEFAULT VALUE

Set to 0 every AddItem()

PURPOSE

Determines the offset from the top of the current item for the next subitem created. This value is incremented automatically every time you call AddSub(). When you allocate a new item, this value is reset to 0.

NAME

nextItemWidth : INTEGER;

DEFAULT VALUE

Set by AddMenu()

PURPOSE

Determines the width of the selection area for the next item created. This value is made equal to the itemWidth parameter every time you call AddMenu().

NAME

nextItemHeight : INTEGER;

DEFAULT VALUE

10

PURPOSE

Determines the height of the selection area for the next item created.

NAME

nextMenuFlags : MenuFlagSet;

DEFAULT VALUE

MenuFlagSet{MenuEnabled}

PURPOSE

Determines the flags used for the next menu created. See the Intuition manual for a definition of all the possible menu flags.

NAME

nextSubFlags : ItemFlagSet;

DEFAULT VALUE

ItemFlagSet{ItemText,ItemEnabled}

PURPOSE

Determines the flags used for the next subitem created. See the Intuition manual for a definition of all the possible subitem flags.

NAME

nextTextAttr : TextAttrPtr;

DEFAULT VALUE

NIL

PURPOSE

Determines the text attributes (font, style, size) of the next Intui-Text created. NIL means that the current default font should be used.

NAME

nextFrontPen : INTEGER;

DEFAULT VALUE

0

PURPOSE

Determines the foreground color used for the next IntuiText created.

NAME

nextBackPen : INTEGER;

DEFAULT VALUE

1

PURPOSE

Determines the background color used for the next IntuiText created.

NAME

nextDrawMode : DrawingModeSet;

DEFAULT VALUE

Jam2

PURPOSE

Determines the drawing mode used for the next IntuiText created.

Procedures

NAME

StartStrip - Prepare for a new menu strip

SYNOPSIS

StartStrip();

PURPOSE

This procedure initializes the global variables of this module to a known state. This must be done before starting to create a menu strip.

NAME

DisposeStrip - Deallocate all memory allocated for a particular menu strip

SYNOPSIS

DisposeStrip(menuStrip:MenuPtr);

menuStrip Menu strip to deallocate, allocated by calls in this module.

PURPOSE

Every time you allocate a menu item, this module allocates a chunk of memory to hold the record that is needed. Once you have finished with a menu strip, it is necessary to return the memory used to the system.

The value you pass this procedure will typically be that of currentStrip.

NAME

AddMenu - Add a menu to the current menu strip

SYNOPSIS

```
AddMenu(menuName:ARRAY OF CHAR;  
        itemWidth:CARDINAL);
```

menuName Title that is to serve as name for this menu.
itemWidth Width of the boxes that will contain the items and
 subitems for this menu.

PURPOSE

Add a new menu to the current menu strip.

The itemWidth value determines the pixel width of the boxes into which the items and subitems of this menu will be drawn into.

NAME

AddItem - Add a menu item to the current menu

SYNOPSIS

```
AddItem(itemName:ARRAY OF CHAR; cmdKey:CHAR);
```

itemName Name of the new item.

cmdKey Amiga key (command key) to be associated with this
 item, or 0C if there is no Amiga key.

PURPOSE

Add a new item to the current menu.

cmdKey indicates which character will be associated with this
menu item to serve as menu shortcut for the item.

NAME

AddSub - Add a menu subitem to the current item

SYNOPSIS

AddSub(subName:ARRAY OF CHAR; cmdKey:CHAR);

subName	Name of the new subitem.
cmdKey	Amiga key (command key) to be associated with this subitem, or 0C if there is no Amiga key.

PURPOSE

Add a new subitem to the current menu item.

cmdKey indicates which character will be associated with this menu subitem to serve as menu shortcut for the subitem.

22. EasyPrintPict

The purpose of the EasyPrintPict module is to provide a single procedure to dump any Amiga bit map to a printer.

The Amiga printer drivers included with Workbench V1.3 provide very high quality graphics dumps on ordinary dot matrix printers. Unfortunately, dumping graphics to the printer requires an understanding of how device IO functions, and in particular, how the printer device functions. EasyPrintPict greatly eases the printing of graphics on the Amiga.

PrintRastPort() will dump the contents of a RastPort to the printer. A RastPort is a high-level display component used by the Amiga to describe a drawing area. Both Intuition windows and screens contain rastports.

Using EasyPrintPict

To print a RastPort, you must:

1. Obtain a RastPort. This may be done by opening a window or screen.
2. Prepare the RastPort by rendering the data you wish to print.
3. Call `PrintRastPort()`.
4. Free the RastPort by closing the screen or window.

Example of usage

```
...  
  
sp:=CreateScreen(320,200,5,"An Easy Screen!");  
                                     (* Open a screen      *)  
IF sp # NIL THEN  
    SetScreenColor(sp,0,15,15,15);    (* Make the screen white*)  
  
    IF PrintRastPort(ADR(sp^.RPort), sp^.VPort.ColorMap,  
                     ViewModeSet{},0,0,320,200) THEN  
        END;  
  
        CloseScreen(sp);              (* Close the screen  *)  
END;  
..
```

Procedures

NAME

PrintRastPort - Dump a rastport to a graphics capable printer

SYNOPSIS

```
PrintRastPort(rp:RastPortPtr; cm:ColorMapPtr;  
              modes:ViewModeSet; x,y,width,height:INTEGER):  
              BOOLEAN;
```

rp	The RastPort to dump.
cm	The ColorMap associated with the RastPort to dump.
modes	This describes the type of screen containing the Rast Port.
x,y	The top left corner within the RastPort of the picture to dump.
width,height	The dimensions of the picture within the RastPort.
result	TRUE if the picture was printed. FALSE if something went wrong and the picture was not printed or only partially printed. This can happen, for example, if the printer device cannot be loaded, or if the printer runs out of paper.

PURPOSE

Queues up a request to the printer device to dump the given rastport to the printer. The print will have the general characteristics specified via Preferences. It will also incorporate the data you pass in as parameters.

The rastport composes most of the Amiga's display elements. Typically, the pointer can be obtained from an open screen or window such as:

```
rp:=ADR(screen^.RPort);  
rp:=window^.RPort;
```

The colormap contains values describing the colors of a screen. This determines how the rastport data will appear. If the printer supports color, the colormap will be used to output the rastport data in the proper colors, otherwise the colormap is used to produce a

grey-scale version of the rastport. Typically, the pointer to the colormap will be obtained from an open screen or window such as:

```
cm:=screen^.VPort.ColorMap;  
cm:=window^.WScreen^.VPort.ColorMap;
```

The modes parameter indicates the type of screen the rastport is being used with. This can indicate HAM, HiRes, or one of the other modes. Typically, this value will be obtained from an open screen or window such as:

```
modes:=screen^.VPort.Modes;  
modes:=window^.WScreen^.VPort.Modes;
```

The return value from this procedure indicates the status of the print operation performed. If the print was successful, TRUE will be returned. If something went wrong while printing, FALSE will be returned. Causes of errors include not enough memory, inability to load the printer device or driver from the Workbench disk, out of paper, no printer connected, or printer not online.

23. EasyProps

The purpose of the EasyProps module is to provide procedures to manipulate Amiga proportional gadgets.

Intuition has a type of gadget known as a proportional gadget which are often used for scrolling graphics or textual information. To use these in your programs (assuming for now vertical scrolling), it is necessary to interpret the relationship between the visible fraction of the display with the size of the gadget, and the top line of the display with the actual position (value) of the gadget's knob. This is best expressed by an example.

Consider a text editor's scroll bar. The scroll bar serves to convey two distinct types of information:

- The size of the scroll bar indicates what percentage of the document is currently visible. That means that if the scroll bar occupies 25% of the defined scroll area, then the part of the text displayed on the screen represents 25% of the entire document.
- The position of the scroll bar within the defined scroll area indicates the position of the text displayed on screen relative to the entire document. That means that if the scroll bar is 47% of the way down relative to the defined scroll area, then the text displayed on the screen is 47% of the total document away, from the top of said document.

Intuition does not provide any high-level routines to control proportional gadgets, only low-level routines. It is not easy to correctly adjust all the necessary variables, keeping into account overflows and such. As proof of this, witness many commercial Amiga programs which incorrectly handle proportional gadgets.

The routines from this module build on top of the low-level Intuition functions to offer added functionality. This will both simplify your

programming task and will ensure that your programs handle prop gadgets the way they were meant to be.

Procedures

NAME

SetProp - Adjust the value of a proportional gadget.

SYNOPSIS

```
SetProp(window      : WindowPtr;  
         gadget      : GadgetPtr;  
         req          : RequesterPtr;  
         totalLines   : LONGCARD;  
         visibleLines : CARDINAL;  
         topLine      : LONGCARD);
```

window	Window containing the proportional gadget.
gadget	Gadget to modify. Must be a PropGadget.
req	Requester containing the gadget. This should be NIL if the gadget is not in a requester.
totalLines	The total number of lines that are available.
visibleLines	The maximum number of lines that can fit in one display.
topLine	The current line at the top of the display. Line numbering starts at 0.

PURPOSE

Adjust the size and position of a proportional gadget given the input values.

totalLines indicates the total number of displayable data elements controlled by this prop gadget. Given the example of a text editor mentioned above, this value would correspond to the total number of lines in the document.

visibleLines indicates how many elements can fit in one view. Given the example of a text editor mentioned above, this would correspond to the total number of lines which fit on screen given the current dimensions of the window.

topLine indicates where in the series of elements, the first visible element exists. Given the example of a text editor mentioned above, this would correspond to the line number of the first line on screen. The first line of the document is line 0.

NAME

GetTopLine - Obtain the 'top line' given the value of a proportional gadget. This is useful after the user has moved the gadget.

SYNOPSIS

```
GetTopLine(gadget      : GadgetPtr;  
           totalLines  : LONGCARD;  
           visibleLines: CARDINAL): LONGCARD;
```

gadget Gadget to read from. Must be a PropGadget.
totalLines The total number of lines that are available.
visibleLines The maximum number of lines that can fit in one display.

result Returns the current top line as specified by the value of the prop gadget.

PURPOSE

Read a prop gadget and determine what is the new top line of a display.

The values of totalLines and visibleLines are as for SetProp().

This routine will typically be called after the user has modified the position of a prop gadget in order to figure out what is the new data to display. Given the example of a text editor as mentioned above, this function would return the line number of the line which should be displayed at the top of the window. The first line of the document is line 0.

24. EasyReadPict

The purpose of the EasyReadPict module is to provide a single procedure to read in an IFF picture from a disk file.

The Interchange File Format (IFF) provides the ability to nest various entities within a larger IFF file. This module allows you to read an InterLeaved Bit-Map (ILBM) as one such entity in a larger encompassing IFF file.

ReadPicture() parses the contents of an ILBM FORM. It calls a parameter-passed procedure to allocate the memory to store the BODY chunk of the ILBM. The BODY chunk is where the actual data of an ILBM is contained. The procedure receives an ILBMFrame record which contains all the data extracted so far from the ILBM FORM. This data contains among other things the size of the BODY (dimensions of the picture). The procedure can simply open a screen of the correct dimensions and let the reading routine fill in the screen with the data from the IFF file.

Using EasyReadPict

To read an IFF format picture, you must:

1. Define a procedure which allocates a bit map.
2. Open a file for Input using BufferedDOS.Open().
3. Call ReadPicture() passing a pointer to your bit map allocation procedure.
4. Close the opened file using BufferedDOS.Close().

Example of usage

```
...

PROCEDURE GetBitMap (VAR frame:ILBMFrame): BitMapPtr;
    BEGIN
        (* open a screen of the correct dimensions *)
    END GetBitMap;
...

file:=Open(ADR("picture.pic"),ModeOldFile); (* Try to open a file*)
IF GoodHandle(file) THEN                    (* Did it open?      *)

IF ReadPicture(file,GetBitMap) THEN
    (* success *)
END;

IF Close(file) THEN END;                    (* Close the file      *)

CloseScreen(sp);                           (* Close the screen    *)
END;
...
```

Variables

NAME

readState : IFFP;

DEFAULT VALUE

None

PURPOSE

The IFF routines provide a standard set of return codes as defined in IFF.def.

Whenever your program reads a picture, this variable will contain the return code obtained from the lower-level IFF routines. If the read operation failed, you can look at this variable to learn more about the cause of the failure.

NAME

currentFrame : ILBMFrame;

DEFAULT VALUE

None

PURPOSE

This variable contains context information relating to the last file that was read. Basically, an ILBM file contains the data for a bit-map, but there are also several additional data chunks which describe the environment in which the bit-map is to appear. Such information includes screen colors, color cycling, screen position of the picture, and other parameters. All this information will be deposited in this variable as it is obtained from the IFF file.

Procedures

NAME

ReadPicture - Read an IFF ILBM from a file

SYNOPSIS

```
ReadPicture(file:FileHandle; getbmap:BITMAPPROC):  
    BOOLEAN;
```

file	A file opened for reading.
getbmap	A procedure to call to get memory to store the data from the file.
result	TRUE if the picture was read, FALSE if something went wrong and the picture was not read or only partially read. Upon failure, the readState variable can be examined to obtain more information on the cause of the failure.

PURPOSE

ReadPicture() takes an IFF ILBM FORM and extracts the data storing the result in an Amiga bit map.

The file parameter can be obtained by opening a file for reading using the BufferedDOS.Open() procedure.

"getbmap" is a procedure that ReadPicture() will call automatically when it requires some memory to read in the BODY chunk of the ILBM FORM. This procedure receives an ILBMFrame as parameter, which it uses to determine the size of the bit map to allocate.

Typically, this routine will open a screen of the size indicated in the BitMapHeader included in the ILBMFrame, and then return a pointer to the screen's bitmap. That way, the reading routine will read the data directly into the screen's bitmap. If there is not enough memory to allocate a bit map, this procedure should return NIL.

Upon successful return from this procedure, the values in the variable "currentFrame" will have been initialized. These values reflect various properties pertaining to the picture just read in, such

as screen color, or color cycling information. You can use these values to determine what to do with the picture. For example, you could use the `colorMap` array to adjust the screen colors in which the picture is displayed.

The return value from this procedure indicates the status of the read operation performed. If the read was successful, `TRUE` will be returned. If something went wrong while reading, `FALSE` will be returned. Some causes of errors include incorrect format of file, no memory, and I/O error.

25. EasySavePict

The purpose of the EasySavePict module is to provide a single procedure to save an Amiga bit map as an IFF picture on disk.

The IFF format provides the ability to nest various entities within a larger IFF file. This module allows you to save an ILBM as one such entity in a larger encompassing IFF file.

Using EasySavePict

To save a picture, you must:

1. Generate a picture.
2. Open a file for output using `BufferedDOS.Open()`.
3. Call `SavePicture()`.
4. Close the opened file using `BufferedDOS.Close()`.

Example of usage

```
...  
  
sp:=CreateScreen(320,200,5,"An Easy Screen!");  
IF sp # NIL THEN  
  
    file:=Open(ADR("picture.pic"),ModeNewFile);  
    IF GoodHandle(file) THEN  
  
        IF SavePicture(file,ADR(sp^.BMap),  
                        sp^.VPort.ColorMap^.ColorTable) THEN  
            (* success *)  
        END;  
  
        IF Close(file) THEN END;    (* Close the file    *)  
    END;  
  
    CloseScreen(sp);                (* Close the screen *)  
END;  
...
```

Variables

NAME

saveState : IFFP;

DEFAULT VALUE

None

PURPOSE

The IFF routines provide a standard set of return codes. These are defined in IFF.def.

Whenever your program saves a picture, this variable will contain the return code obtained from the lower-level IFF routines. If the save operation failed, your program can look at this variable to learn more about the cause of the failure.

Procedures

NAME

SavePicture - Save a picture into an IFF file

SYNOPSIS

```
SavePicture(file:FileHandle; bMap:BitMapPtr;  
            colors:ColorTablePtr) : BOOLEAN;
```

file	A file opened for writing.
bMap	The bitmap containing the data to save as an ILBM.
colors	The colors associated with the bitmap (may be NIL).
result	TRUE if the picture was saved. FALSE if something went wrong and the picture was not saved or only partially saved. Upon failure, the saveState variable can be examined for more information on the cause of the failure.

PURPOSE

SavePicture() takes an Amiga bit map and saves its contents as an IFF ILBM FORM. The FORM is output in the file as is. This means that the file may be a larger IFF file being written which is composed of several FORMs ILBM.

The file parameter can be obtained by opening a file for writing using the BufferedDOS.Open() procedure.

The BitMap is the most primitive display element on the Amiga. It describes regions of memory arranged in bit-planes. Typically, this pointer will be obtained from an open screen such as:
bMap:=ADR(screen^.BMap);

A color table is a set of 16-bit values that describe the contents of the Amiga's color registers when displaying a given bit map. If there are no specific colors, NIL can be passed for this parameter. Typically, this pointer will be obtained from an open screen or window such as:

```
colors:=ADR(screen^.VPort.ColorMap^.ColorTable)  
coors:=ADR(window^.WScreen^.VPort.ColorMap^.ColorTable)
```

The return value from this procedure indicates the status of the save operation performed. If the save was successful, TRUE will be returned. If something went wrong while saving, FALSE will be returned. Some causes of errors are disk full, and I/O error.

26. EasyScreens

The purpose of the EasyScreens module is to provide procedures to create and maintain Amiga screens.

Intuition provides several procedures to manipulate screens. In order to create a screen with Intuition, a complex structure must be initialized and passed to Intuition for further processing. If there is enough memory, a screen will be created.

This scheme works fine as long as the NewScreen record has been properly initialized before it is given to Intuition. If some fields are not properly initialized, your program will crash, and finding the problem can be difficult. As well, to write small test programs, it is tedious to constantly have to initialize those screen structures. EasyScreens provides a single procedure to initialize all the necessary variables and then call the Intuition OpenScreen() procedure.

Using EasyScreens

To open and use a screen you must:

1. Call `CreateScreen()` to actually open the screen.

Before opening the screen, you may wish to modify the nextXXX variables in order to alter the characteristics of the screen.

2. Manipulate the screen using the standard Intuition calls.
3. Call `Intuition.CloseScreen()` to close the screen and free the memory.

Example of usage

...

```
sp:=CreateScreen(320,200,5,"An Easy Screen!");
                                (* Try to open a screen      *)
IF sp # NIL THEN                (* Make sure it opened       *)
    SetScreenColor(sp,1,15,0,0);
                                (* Make background red       *)
    Delay(100);                 (* Wait 2 seconds      *)

    SetScreenColor(sp,1,0,15,0);
                                (* Make background green    *)
    Delay(100);                 (* Wait 2 seconds      *)

    SetScreenColor(sp,1,0,0,15);
                                (* Make background blue     *)
    Delay(100);                 (* Wait 2 seconds      *)

    CloseScreen(sp);            (* Close the screen    *)
END;
```

...

Variables

NAME

nextDetailPen : CARDINAL;

DEFAULT VALUE

0

PURPOSE

This variable defines which pen number will be used to render the title of the next screen opened.

NAME

nextBlockPen : CARDINAL;

DEFAULT VALUE

1

PURPOSE

This variable defines the pen number used to render the title bar of the next screen opened.

NAME

`nextModes : ViewModeSet;`

DEFAULT VALUE

`ViewModeSet{}`. This means low-resolution display, non-interlace.

PURPOSE

This variable defines the modes of the next screen to be opened.

Modes indicate how the hardware is to display the screen data. You can specify HiRes, Lace, GenLock, etc...

NAME

`nextFlags : ScreenFlagSet;`

DEFAULT VALUE

`ScreenFlagSet{}`

PURPOSE

This variable defines several flags that will be used when opening the next screen. Your program can specify `ScreenBehind` to force the screen to initially open in the back of all screens (you can then bring it to the front using the `Intuition.ScreenToFront()` procedure), or specify `ScreenQuiet` so that the screen won't have a title bar rendered.

NAME

`nextTextAttr : TextAttrPtr;`

DEFAULT VALUE

NIL

PURPOSE

This variable defines the default font to use when opening the next screen. If this parameter is NIL, then the current default system font will be used. If you specify a value for this variable, the screen title, the window titles and the menu titles will all be rendered in this font. Note that the font must already be in memory when opening the screen; i.e. Intuition will not load fonts from disk.

Procedures

NAME

CreateScreen - Open a screen

SYNOPSIS

```
CreateScreen(width,height,depth:INTEGER;  
              title:ARRAY OF CHAR): ScreenPtr;
```

width The screen's width in pixels.
height The screen's height in pixels.
depth The number of bit-planes the screen should have.
title The screen's title.

result A pointer to the screen, or NIL if it could not open.

PURPOSE

Open an Intuition screen. The screen has the characteristics described by the "nextXXX" variables. The return value from this procedure points to the new screen structure, or will be NIL if the screen could not be opened (probably due to lack of memory).

If you wish your software to support various video standards (NTSC or PAL), the height parameter should be set to Intuition.StdScreenHeight (-1).

Current Amiga hardware supports from 1 to 6 bit-planes depending on the resolution.

The screen can be closed by calling the Intuition.CloseScreen() procedure.

NAME

SetScreenColors - Adjust the display colors of a screen.

SYNOPSIS

```
SetScreenColor(screen:ScreenPtr; colorNumber:CARDINAL;  
               red,green,blue:CARDINAL);
```

screen	Screen for which to set the colors.
colorNumber	The pen number whose color is being changed.
red,green,blue	New color values for the given colorNumber.

PURPOSE

Allows you to easily change the color values for any given pen number. Each Amiga color has a red, green, and blue value that can range from 0 (no color) to 15 (full color).

27. EasySpeech

The purpose of the EasySpeech module is to provide procedures to control the Amiga's narrator device and translator library in order to easily generate speech.

The Amiga is the only personal computer available today that has built-in speech capabilities. Speech can be very useful to enhance a variety of applications. The Amiga's OS provides the "translator" library to convert english text strings into their phoneme equivalents. The "narrator" device is then used to speak these phonemes. This is a flexible approach which allows one to replace the translator library in order to translate any language into phonemes.

Unfortunately, using the narrator device requires an understanding of how device IO functions, and in particular, how the narrator device functions. EasySpeech greatly eases the task of making the Amiga talk. It will take english strings and convert them into audio output.

Using EasySpeech

To have the Amiga speak a string:

1. Adjust the "nextXXX" (rate, pitch, sex) variables to the desired values for the speech.
2. Use the Say() or SayAndReturn() procedures to make the Amiga speak.

Example of usage

```
...  
nextMouth:=TRUE;          (* Generate Mouth reports      *)  
IF NOT speechFailed THEN  (* Was everything setup?    *)  
  
    SayAndReturn("Hello world!");  
                        (* Just start talking              *)  
  
    IF ReadMouth(width,height) THEN  
        (* Get mouth dimensions                            *)  
        WriteCard(width,5);  
        WriteCard(height,5); WriteLn;  
    END;  
  
END;  
...
```

Variables

NAME

speechFailed : BOOLEAN;

DEFAULT VALUE

None

PURPOSE

This variable holds state information relating to the last speech operation performed. If the last operation failed, this value will be FALSE, otherwise it will be TRUE. Possible cause of errors include lack of memory, inability to obtain signal bits, and the inability to load the narrator device or the translator library from disk.

NAME

currentPhoneme : ARRAY [0..256] OF CHAR;

DEFAULT VALUE

None

PURPOSE

To have the narrator device 'talk a string', this string must first be transformed in phonemes. This transformation is performed by the translator library. The routines from this module use this variable to store the phoneme strings that the narrator device is to manipulate. It may be useful to know exactly what the narrator device is saying.

NAME

nextVolume : VolRange;

DEFAULT VALUE

NarratorDevice.DefVol (64)

PURPOSE

This variable determines the volume to use when generating the next speech output. This value is in the range [0..64], 64 being the loudest volume possible.

NAME

nextSampFreq : FreqRange;

DEFAULT VALUE

NarratorDevice.DefFreq (22200 Hz)

PURPOSE

This variable determines the frequency of the audio output. Higher frequencies ensure better sound quality, but take more CPU time to generate. This value is in the range [5000..28000] and represents Hertz.

NAME

nextRate : RateRange;

DEFAULT VALUE

NarratorDevice.DefRate (150 words/minute)

PURPOSE

This variable determines the speaking rate used. The higher the value, the faster the output is. This value is in the range [40..400] and represents an average number of words per minute.

NAME

nextPitch : PitchRange;

DEFAULT VALUE

NarratorDevice.DefPitch (110 Hz)

PURPOSE

This variable determines the pitch of the audio output. The higher the value, the higher the pitch. This value is in the range [65..320] and represents Hertz.

NAME

nextSex : SexRange;

DEFAULT VALUE

NarratorDevice.DefSex (Male (0))

PURPOSE

This variable determines the sex used when talking. The narrator device will simulate different inflections in the voice depending on which sex is used. This value can be either Male (0) or Female (1).

NAME

nextMode : ModeRange;

DEFAULT VALUE

NarratorDevice.DefMode (Natural (0))

PURPOSE

This variable determines the type of voice used when generating speech. There are two types of speech: Natural and Robotic. When using Natural speech, the narrator device will try to provide normal human intonations in the speech. If Robotic is used, no such intonations will be provided. Using the Robotic voice may result in better quality speech.

NAME

nextMouth : BOOLEAN;

DEFAULT VALUE

FALSE

PURPOSE

The narrator device can generate width and height values representing a mouth which would be saying what the narrator device is saying. Generating these values takes some CPU time. By setting this value to TRUE, the narrator device will generate the mouth values, otherwise it will not. This mouth feature may be useful in generating video images that are synchronized with the audio output to produce the illusion that the video images are talking.

Procedures

NAME

ReadMouth - Read width & height values to simulate a mouth while the Amiga speaks.

SYNOPSIS

ReadMouth(VAR width,height:CARDINAL): BOOLEAN;

width,height Variables to receive the current dimensions of a virtual mouth speaking the text that the Amiga is currently narrating.

result TRUE if everything OK, or FALSE if there is currently no speech in progress or if there was some internal error.

PURPOSE

This procedure is called to read width and height values for a mouth that would be saying what the Amiga is currently saying. This is useful in generating 'talking' video images.

The two parameters receive the width and height values.

The return value tells you if the call succeeded. If this value is TRUE, then the values in width and height are valid. Otherwise, the values are undefined. The most probable cause of error is trying to read when there is no speech in progress.

NAME

Say - Make the Amiga talk

SYNOPSIS

Say(text:ARRAY OF CHAR);

text English string for the Amiga to speak.

PURPOSE

This procedure is the most important one of this module. This is the procedure that makes your Amiga talk.

You simply pass this routine a string of characters and the Amiga will start speaking it. This procedure waits for the speech to be finished before returning.

Upon return, you can look at the speechFailed variable to determine if the speech output was successful or not.

NAME

SayAndReturn - Make the Amiga talk

SYNOPSIS

SayAndReturn(text:ARRAY OF CHAR);

text English string for the Amiga to speak.

PURPOSE

This is a variation of the Say() procedure defined above. To use this procedure, you simply pass it a string of characters that the Amiga should say.

The difference between this procedure and the normal Say() procedure is that this procedure does not wait for the speech to be finished before returning. This means that your program can continue executing while the speech is in progress.

If there is some speech in progress and you try to issue another speech command, your program will be blocked until the first speech request is complete and the new one is started.

NAME

StopSpeech - Abort any speech started by SayAndReturn()

SYNOPSIS

StopSpeech();

PURPOSE

This procedure aborts any outstanding speech that was started by SayAndReturn().

NAME

WaitSpeech - Wait for any speech started by SayAndReturn() to be finished

SYNOPSIS

WaitSpeech();

PURPOSE

This procedure waits for any outstanding speech request started by SayAndReturn() to complete. This may be very useful when trying to synchronize your program with speech output.

28. EasyWindows

The purpose of the EasyWindows module is to provide a procedure to create Amiga windows.

Intuition provides several procedures to manipulate windows. In order to create a window, a complex structure must be initialized and passed to Intuition for further processing. If there is enough memory, a window will be created.

This scheme works fine as long as the NewWindow record is properly initialized before being passed to Intuition. If your program does not initialize some fields, it will crash, and finding the problem can be difficult. In addition, when writing "quick and dirty" test programs, it can be tedious to constantly have to write the code to initialize the window structures. EasyWindows provides a single procedure to initialize all the necessary variables and then call the Intuition `OpenWindow()` procedure.

Using EasyWindows

To open and use a window you must:

1. Call CreateWindow() to actually open the window

Before opening the window, you may wish to modify the nextXXX variables in order to alter the characteristics of the window.

2. Manipulate the window using the standard Intuition calls.

3. Call Intuition.CloseWindow() to close the window and free the memory.

Example of usage

...

```
wp:=CreateWindow(10,10,350,75,"An Easy Window!",
                IDCMPFlagSet{},
                WindowFlagSet{WindowDrag,
                              WindowSizing},NIL,NIL);
```

```
IF wp # NIL THEN                                (* Make sure it opened *)
    Delay(200);                                  (* Wait 4 seconds      *)
    CloseWindow(wp);                             (* Close the window   *)
END;
...
```

Variables

NAME

nextDetailPen : CARDINAL;

DEFAULT VALUE

0

PURPOSE

This variable defines which pen number will be used to render the title of the next window opened. It also defines the color used for the rendering of menu titles.

NAME

nextBlockPen : CARDINAL;

DEFAULT VALUE

1

PURPOSE

This variable defines which pen number will be used to render the borders of the next window opened. It also defines the color of the background of menu strips attached to this window.

Procedures

NAME

CreateWindow - Open a window

SYNOPSIS

```
CreateWindow(leftEdge,topEdge,width,height:INTEGER;  
             title:ARRAY OF CHAR; idcmp:IDCMPFlagSet;  
             flags:WindowFlagSet; screen:ScreenPtr;  
             gadgets:GadgetPtr) : WindowPtr;
```

leftEdge	The window's left edge in pixels from left of screen.
topEdge	The window's top edge in pixels from top of screen.
width	The window's width in pixels.
height	The window's height in pixels.
title	The window's title.
idcmp	The window's IDCMP flags. Used to receive events from the window. See EasyIDCMP.
flags	Controls what type of window is opened, and what system gadgets are attached to this window. See Intuition.def for the various flag definitions.
screen	A pointer to a screen in which this window should open. If this value is NIL, the window will open in the Workbench screen.
gadgets	Pointer to the first of a list of gadgets that should be attached to this window when the window is first opened. This value may be NIL, in which case no gadgets will be attached.
result	A pointer to the window, or NIL if it could not open.

PURPOSE

Open an Intuition window. The window will have the characteristics described by the "nextXXX" variables. The return value from this procedure points to the new window structure, or will be NIL if the window could not be opened (probably due to lack of memory).

You close the window by calling the Intuition.CloseWindow() procedure.

1

2

3

A. Error messages and return codes

Compiler errors

The following is a list of M2C's compiler errors with error number, description, and extended explanation.

10 identifier expected

The compiler expected an identifier, but could not find it. An identifier starts with a letter, which can be followed by either letters or numbers. Two identifiers spelled the same, but using different letter case (i.e. Hello and HELLO) are considered to be different. Modula-2 keywords cannot be used as identifiers (such as BEGIN, PROCEDURE, etc...)

11 , comma expected

The compiler expected a comma, but could not find it. Commas are required to separate import list elements, set elements, case labels, enumeration type declarations and variable declarations.

12 ; semicolon expected

The compiler expected a semicolon, but could not find it. Semicolons are used to separate every Modula-2 statement.

13 : colon expected

The compiler expected a colon, but could not find it. Colons are used in variable declarations to separate the variable names from their types. They are also used in CASE constructs to separate the labels from the code associated with the labels.

14 . period expected

The compiler expected a period, but could not find it. Every Modula-2 compilation unit must terminate with a period.

15) right parenthesis expected

The compiler expected a parenthesis, but could not find it. There always has to be an even number of parenthesis in a Modula-2 statement. This means that every left parenthesis must have a matching right parenthesis. Parenthesis can be used in procedure calls and to provide clarity in expression evaluation.

16] right bracket expected

The compiler expected a bracket, but could not find it. There always has to be an even number of brackets in a Modula-2 statement. This means that every left bracket must have a matching right bracket. Brackets are used in declaring and using arrays.

17 } right brace expected

The compiler expected a brace, but could not find it. There always has to be an even number of braces in a Modula-2 statement. This means that every left brace must have a matching right brace. Braces are used when referring to sets.

18 = equal sign expected

The compiler expected an equal sign, but could not find it. This error will appear in a TYPE or CONST declaration block, when an identifier is not made equal to something.

19 := assignment expected

The compiler expected an assignment operator, but could not find it.

20 END expected

The compiler expected an END statement, but could not find it. Every complex Modula-2 construct requires an END statement to

indicate it is complete. If an END statement is forgotten, the compiler will report this error at the end of the current procedure or module body. When this occurs, it must be verified that every construct in the current procedure has a corresponding END statement.

21 .. ellipsis expected

The compiler expected an ellipsis, but could not find it. Ellipsis are used to declare subranges of values.

22 (left parenthesis expected

The compiler expected a left parenthesis, but could not find it. This error will occur when calling a procedure which requires parameters. If no parameters are supplied to the procedure when it expects some, or if the parameter list starts without having the required parenthesis before them.

23 OF expected

The compiler expected an OF statement, but could not find it. OF is used in array declarations, set declarations and CASE constructs.

24 TO expected

The compiler expected a TO statement, but could not find it. TO is used in FOR loop constructs to indicate the upper bound of the loop or after the word POINTER in a type declaration.

25 DO expected

The compiler expected a DO statement, but could not find it. DO is used in FOR loops and in WHILE loops. DO indicates the end of the loop declaration and the start of the loop body. This error will occur if the DO statement is omitted, or if the loop declaration is malformed.

26 UNTIL expected

The compiler expected an UNTIL statement, but could not find it.

UNTIL statements are used to indicate the end of a REPEAT loop. If an UNTIL statement is forgotten, the compiler will report this error at the end of the current procedure or module body. When this occurs, it must be verified that every REPEAT has a matching UNTIL statement within the current procedure.

27 THEN expected

The compiler expected a THEN statement, but could not find it. THEN is used in IF or ELSIF constructs. This error will occur if the THEN statement is omitted or if the boolean expression forming the condition to test is malformed.

28 MODULE expected

The compiler expected a MODULE statement, but could not find it. Every Modula-2 compilation unit has to have the word MODULE in its heading.

29 illegal digit, or number too large

A numeric constant has been improperly declared. Either the number contains illegal characters, or the number is too large to represent in the current type.

30 IMPORT expected

The compiler expected an IMPORT statement, but could not find it. IMPORT is used in a module's import list. This error will occur if the IMPORT statement is omitted or misspelled. Note that module names cannot have spaces in them. For example, the following is invalid:

```
FROM Super Stuff IMPORT ...;
```

Such a construct would yield this error.

31 factor starts with illegal symbol

An illegal symbol was found where an expression should start.

32 identifier, (, or [expected

The compiler expected the start of a simple type but could not find it. A simple type can be a type identifier, an array declaration or an enumeration type declaration.

33 identifier, ARRAY, RECORD, SET, POINTER, PROCEDURE, (, or [expected

The compiler expected a type specification, but could not find it. This error will occur when a type is being defined, but no value is associated to it, such as:

TYPE

List = ; (* Must be a definition here *)

34 type followed by illegal symbol

The compiler encountered invalid characters after the end of a type declaration. This error will occur if you forget a semicolon, or an END statement at the end of a type definition.

35 statement starts with illegal symbol

The compiler expected a statement, but could not find it.

36 declaration followed by illegal symbol

The compiler expected either of the following: BEGIN, CONST, END, MODULE, PROCEDURE, TYPE, VAR, but could not find it.

37 statement part is not allowed in DEFINITION

The compiler found some implementation details in a definition module. Definition modules only hold constants, type, variables and the heading of procedures. The actual code to implement the procedures is found in the associated implementation module.

38 export list not allowed in IMPLEMENTATION module

An attempt was made to declare an export list in an implementa-

tion or program module. Export lists are only allowed in definition modules.

39 EXIT not inside a LOOP construct

An attempt was made to use the EXIT statement while there is no active LOOP construct. EXIT is only allowed within a LOOP..END pair.

40 illegal character in number

An illegal character was encountered in the processing of the specified number.

41 number too large

The value of the literal or computed value exceeds the limits of the current type.

42 comment without closing *)

A comment section was opened, but never closed. When a comment is not closed, it will cause the compiler to think that all the source code from the open comment onto the end of the source file is part of the comment.

44 expression must contain constant operands only

The given expression must not contain any variables or procedure calls, only literals expressions.

45 control character within string (cannot exceed one line)

A Modula-2 string literal must be contained entirely on one program line, it cannot be split onto two or more lines.

50 identifier not declared or not visible

An attempt was made to reference an identifier which was never declared. This implementation requires that all identifiers be declared textually before they are first used. This error will often occur because of a typo.

- 51 object should be a constant

The compiler requires a constant, but the specified object is not one.

- 52 object should be a type

The compiler requires a type identifier, but the specified object is not one.

- 53 object should be a variable

The compiler requires a variable, but the specified object is not one.

- 54 object should be a procedure

The compiler requires a procedure, but the specified object is not one.

- 55 object should be a module

The compiler requires a module, but the specified object is not one.

- 56 type should be a subrange

The compiler requires a subrange type, but the specified object is not one.

- 57 type should be a record

The compiler requires a record type, but the specified object is not one.

- 58 type should be an array

The compiler requires an array type, but the specified object is not one.

59 type should be a set

The compiler requires a set type, but the specified object is not one.

60 illegal base type of set

A set must be formed from either a subrange or an enumeration.

61 incompatible type of CASE label or of subrange bound

All the labels specified in a CASE construct must be of the same type as the CASE expression. For example, if you have the following:

```
VAR
  x : CARDINAL;
...

CASE x OF
  (* CASE expression is a
    * CARDINAL *)
  | 0 : ...; (* This is OK *)
  | 1 : ...; (* This is OK *)
  | "A": ...; (* Label is not a
    * CARDINAL,error *)
```

62 CASE label already defined

In a CASE construct, every label must be unique. This also includes labels contained within a subrange. For example:

```
CASE x OF
  | 0..9: ...; (* All values from 0 to 9 *)
  | 5 : ...; (* Illegal, 5 is already in
    * the above subrange *)
```

63 low bound > high bound

An attempt was made to declare a subrange with the upper bound being smaller than the lower bound. For example:

```
ARRAY [100..0] OF CHAR;
```

```
CASE x OF  
  |10..5: ;  
END;
```

64 too many parameters specified

An attempt was made to pass more parameters to a procedure, than the procedure actually requires. Verify the definition of the procedure to validate your parameters.

65 more parameters expected

The specified procedure requires more parameters than were supplied. Verify the definition of the procedure to learn what parameters are missing

66 more parameters in IMP than in DEF

The implementation of a procedure has more parameters than its definition found in the definition module.

67 parms with equal types in IMP have different types in DEF

Parameters of a procedure in an implementation module have differing types, while they were declared as being the same in the definition module.

68 mismatch between VAR specifications

The specified parameter is declared differently in the definition module than in the implementation module. Either the definition specifies the parameter as being a VAR parameter and the implementation doesn't, or vice-versa.

69 mismatch between parameter type specifications

The specified parameter has a different type in the implementation than in the definition.

70 more parameters in DEFINITION than in IMPLEMENTATION

The implementation of a procedure has fewer parameters than its definition found in the definition module.

71 mismatch between result type specifications

The type of the return value of the procedure in the implementation differs from the type of the return value of the procedure in the definition.

72 procedure in DEF returns value, but not in IMPLEMENTATION

The definition module defines that a procedure returns a value, while the implementation does not.

73 procedure in DEF has parameters, but not in IMPLEMENTATION

The definition module defines that this procedure expects parameters while the implementation does not.

75 illegal type of control variable in FOR construct

The control variable in a FOR loop can't be REAL, LONGCARD or LONGINT. The variable must also be a simple variable, it cannot be part of a record.

76 procedure call returns a value

The specified procedure returns a value, but an attempt was made to call it as if it didn't. Verify the definition of the procedure to obtain the return value.

77 identifiers in heading and at end do not match

The name of a procedure or module differs from the identifiers specified at the END of the procedure or module. For example:

```
PROCEDURE Test;  
BEGIN  
  ...  
END TestCode; (* Should be Test *)
```

This error will also occur if an excess END is encountered in a procedure or module body.

78 type already declared in DEFINITION

An attempt was made to redefine a type in an implementation module while it is already defined in the definition module.

79 imported module not found

The specified module was not found on disk.

80 unsatisfied export list entry

An entry specified in the export list was not declared in the definition module. You should either remove the entry from the export list, or define it properly in the current module.

81 illegal type of procedure result

An attempt was made to return a complex value as result from a procedure. This implementation restricts the return values from procedures to be either 1, 2, 4 or 8 bytes. The size of an identifier can be determined by using the TSIZE procedure. Larger types should be transferred from a procedure using a VAR parameter.

82 illegal base type of subrange

The specified type is not valid as the base type of a subrange.

83 illegal type of CASE expression

The expression to evaluate in a CASE construct produces a result which is incorrect for a CASE expression. CASE expressions are limited to using 16 bit values. This means that LONGCARD, LONGINT, REAL and LONGREAL are not allowed.

84 symbol file not successfully written (disk full?)

The compiler could not complete it's output of a symbol file. Most probable that there was either a disk error, or the disk was

full.

85 keys of imported symbol files do not match

There is a version conflict between the module specified and some other modules that were imported. This means a definition module was recompiled, while some modules that depended on it were not. The best solution to this problem is to recompile the entire set of modules from scratch.

86 error in format of symbol file

There is an inconsistency in the format of the specified symbol file. This is probably due to a disk error.

87 unresolved pointer type in current block

A pointer type has been defined to an object, but said object has not been defined within the current block. For example:

TYPE

```
Ptr = POINTER TO Object;  
(* And if the type 'Object' is never  
   * defined, this error occurs      *)
```

88 symbol file not successfully opened

The specified symbol file could not be opened for input. There was either a disk error or a lack of memory.

89 procedure declared in DEF, but not in IMPLEMENTATION

A procedure was defined in the definition module, but was never implemented in the implementation module.

90 in {a..b}, if a is a constant, b must also be a constant

When accessing a set, all accesses must be of a similar type within any given expression. This means that if you access a set element directly by using a constant, all other set elements must also be specified as constants.

- 92 too many cases (more than 128)

In a CASE construct, or in a record variant, there may not be more than 128 different cases.

- 93 too many EXIT statements (more than 16)

There may not be more than 16 EXITS for any given LOOP construct.

- 94 index type of array must be a subrange

The type used to index an array must be a range or subrange, that is the type must have a lower and upper limit that describes the range of the array.

- 95 subrange bound must be less than 2^{15}

The upper limit of the subrange must be less than 2 to the power 15 (32768).

- 96 too many global modules

Implementation restriction. You have attempted to import more modules than the compiler can handle.

- 98 too many structure elements in DEFINITION module

Implementation restriction. You have imported modules that contain too many structures (types, vars, procedures etc) for the compiler to handle.

- 100 multiple definitions within the same scope

Two or more items in the current scope have the same name.

- 107 illegal use of module

You have used a module name where it is not expected. Module names can only be used in IMPORT statements, or in qualified import.

108 constant index out of range

The array index specified exceeds the declared bounds of the array.

109 indexed variable not an array, or index has wrong type

An attempt has been made to index an item that is not an array, or an incorrect index type has been used to index an array (eg. you can't index an array by a REAL number).

110 record selector is not a field identifier

The given identifier is not a field of the record being accessed.
For example:

```
VAR
  x : RECORD
    a,b : INTEGER;
  END;
BEGIN
  x.c:=2; (* c is not part of the record *)
```

111 dereferenced variable is not a pointer

An attempt was made to use the dereferencing arrow (^) on an object which is not a pointer. For example, you may not dereference a procedure, a constant or type. As well, the following is invalid:

```
VAR
  x : CARDINAL;
BEGIN
  x^:=2;
```

112 operand type incompatible with sign inversion

You have attempted to invert the sign of a type that is not signed.
For example;

```

VAR
  c : CARDINAL ;      (* is not signed *)
BEGIN
  c := -c ;           (* is illegal      *)

```

113 operand type incompatible with NOT (~)

Only BOOLEAN type may be used with NOT.

114 x IN y; type(x) # basetype(y)

It is illegal to test a set with anything other than the base type of the set. For example;

```

TYPE
  t : SET OF [0..7];  (* base type is a
                       * number      *)
VAR
  s : t;
BEGIN
  IF 0 IN s THEN (* this is legal *)
END;
  IF 'A' IN s THEN (* this is not legal 'A'
                   * is CHAR not a number*)
END;

```

115 x IN y; type of x cannot be the basetype of a set, or y is not a set

Either the referenced variable is not a set or the referencing item is of the wrong type.

116 {a..b}; type of a or b # of the base type of the set

The types of both the upper (b) and lower (a) bounds of the set constant must be a subrange of the base type of the set.

117 incompatible operand types

The requested operation cannot be performed between the type item types. For example, you cannot have 0+'A' as numbers and characters are not compatible.

118 operand type incompatible with *

The operand type cannot be multiplied (eg. FALSE * 10)

119 operand type incompatible with /

The operand type cannot be divided (ie. is not REAL or LONGREAL). Note the '/' operator is only used with real numbers, other types use 'DIV'.

120 operand type incompatible with DIV

The operand type cannot be divided (eg. 'A' DIV 2)

121 operand type incompatible with MOD

The modulo of the operand cannot be performed (eg. TRUE MOD 'a')

122 operand type incompatible with AND (&)

Only boolean types can be used with AND.

123 operand type incompatible with +

The addition of these operands cannot be performed (eg. 'A'+2)

124 operand type incompatible with -

The subtraction of these operands cannot be performed (eg. 'A'-2)

125 operand type incompatible with OR

Only boolean types can be used with OR.

126 operand type incompatible with relation

Only simple types (no records, arrays or sets) may be used with relational operators.

127 procedure must have level 0, it cannot be nested

In order to pass a procedure as parameter or get the storage address of a procedure, the procedure must not be embedded within order procedure, it must be at the outermost level in a module.

128 incompatible procedure types: return value type mismatch

The specified procedure can't be used in this context. The type of its return value differs from what is required.

129 incompatible procedure types: parameter type mismatch

The specified procedure can't be used in this context. The type of one or more of its parameters differs from what is required.

130 incompatible procedure types: too few parameters

The specified procedure can't be used in this context. The procedure has fewer parameters than what is required.

131 incompatible procedure types: too many parameters

The specified procedure can't be used in this context. The procedure has more parameters than what is required.

132 assignment of a negative integer to a cardinal variable

CARDINAL and LONGCARD, as well as subranges of such, only accept positive values.

133 incompatible assignment

A value or expression must be of the same type as a variable to assign it to the variable. For example, you cannot assign a REAL value to a CARDINAL variable.

134 assignment to non-variable

An attempt was made to assign a value to something which is not a procedure. For example, you may not assign a value to a procedure.

dure, constant or type.

135 type of exp. in IF, WHILE, UNTIL clause must be BOOLEAN

The value or expression used as condition statement must be of type BOOLEAN, which represents either TRUE or FALSE. It is incorrect to use these conditional statements with any other type.

136 call of an object which is not a procedure

An attempt was made to use something which is not a procedure as if it was. For example, you may not call a type.

137 type of VAR parameter differs from procedure specification

The specified parameter does not match what is expected. Verify the definition of the procedure to determine the correct type. Note that this implementation uses ARRAY OF BYTE instead of ARRAY OF WORD as universal open array parameter. This is because the smallest addressable unit on a 68000-family processor is a byte, not a word.

139 type of RETURN expression differs from procedure type

The type of a value returned by a RETURN statement does not match the type of the return value declared in the procedure heading.

141 step in FOR construct cannot be 0

The value used in the BY clause of a FOR construct may not be 0. If it was zero, then an infinite loop would occur, in which case the LOOP..END construct should be used instead.

142 illegal type of control variable

The control variable of a FOR loop must be either of type enumeration, BOOLEAN, CARDINAL, CHAR, INTEGER.

144 incorrect type of parameter of standard procedure

The standard Modula-2 procedure expects a parameter of a differ-

ent type.

145 parameter should be a type identifier

Some standard Modula-2 procedures such as MAX and TSIZE expect type identifiers (CARDINAL, REAL, etc..) as parameters and not expressions.

146 string too long

The specified string literal is too long to assign to the given variable.

147 incorrect priority specification

A module priority should be in the range 0 to 15. Priorities are ignored in this implementation.

148 illegal redefinition of reserved word

Modula-2 keywords cannot be redefined in user programs.

200 implementation restriction

An internal compiler error occurred. This error should not occur. If it does occur, please preserve the current compilation unit and contact the M2S office nearest you to report the error.

201 integer too small for sign inversion

An attempt was made to invert the sign of the smallest INTEGER or LONGINT value possible. For example, the smallest integer value possible is -32768, while the largest is 32767. This obviously means that -32768 can't be sign-inverted.

202 element outside of set range

The element is either not in the range of the set or the set contains more than 32 items.

203 overflow in multiplication

An attempt was made to multiply two constant operands, and the result overflowed the current type.

204 overflow in division

An attempt was made to divide two constant operands, and the result overflowed the current type.

205 division by zero, or modulus with negative value

An attempt was made to perform a division by zero, or the right operand of a modulus operation is negative.

206 overflow in addition

An attempt was made to add two constant operands, and the result overflowed the current type.

207 overflow in subtraction

An attempt was made to subtract two constant operands, and the result overflowed the current type.

208 cardinal value assigned to integer variable too large

An attempt was made to assign a constant which exceeds MAX(INTEGER) to an integer variable. This implementation has MAX(INTEGER) equal to 32767.

209 set size too large

This implementation restricts the maximum number of elements in a compiler set to 32. To use larger sets, see the LargeSets module supplied with the compiler.

210 array size too large

There was an attempt to declare an array which total size is larger than 32K. The total size of all array elements must be under 32K.

211 too much data allocated

There was an attempt to allocate more local data than the compiler allows. There may not be more than 32K of global module data, as well as 32K of local storage for every procedure.

212 attempt to assign string literal to VAR parameter

You cannot assign a string literal to a variable parameter. For example, given the procedure:

```
PROCEDURE Process (VAR str:ARRAY OF CHAR);
```

you cannot call this procedure like:

```
Process ("Hello");
```

You must pass a variable instead.

214 set elements must be constants

The declaration of a set must contain only constant operands, no types or variable names.

215 expression too complex (stack overflow)

The compiler was unable to handle the current expression. To solve the problem, you must split the complex expression into two or more simpler expressions.

222 output file not opened (disk full?)

The compiler was unable to open an output file. This is normally due to a disk error.

223 output incomplete (disk full?)

The compiler could not finish outputting a file. This is normally due to a disk error.

224 too many external references

This error occurs when the compiler's relocation buffer overflows. The problem can usually be solved by increasing the size of the buffer. The relocation buffer is used to store information on every reference to an object which is external to the current compilation unit.

225 too many strings

This error occurs when the compiler's const buffer overflows. The problem can usually be solved by increasing the size of the buffer. The const buffer is used to store information on string literals.

226 code buffer overflow

This error occurs when the compiler's code buffer overflows. The problem can usually be solved by increasing the size of the buffer. If the problem persists, you will have to split the current module into smaller modules.

227 identifier buffer overflow

This error occurs when the compiler's identifier buffer overflows. The problem can usually be solved by increasing the size of the buffer. The identifier buffer is used to store information on every identifier either declared or imported into a module.

228 relocation buffer overflow

This error occurs when the compiler's relocation buffer overflows. The problem can usually be solved by increasing the size of the buffer. The relocation buffer is used to store information on every reference to an object which is external to the current compilation unit.

230 expression not loadable (implementation restriction)

An internal compiler error occurred in a redundancy check.

231 expression not addressable (implementation restriction)

An attempt is being made to get the storage address of something

which does not have an address. For example, trying to use the `ADR()` procedure on a type, constant or literal. An exception to this is that this implementation allows taking the address of string constants and literals.

232 expression not allowed (implementation restriction)

An internal compiler error occurred in a redundancy check.

234 register reservation error

An internal compiler error occurred. This error should not occur. If it does occur, please preserve the current compilation unit and contact the M2S office nearest you to report the error.

235 illegal selector for constant index or field

Code generation error.

236 too many nested WITH statements (more than 4)

There cannot be more than four nested WITH statements at any one time.

237 illegal operand

Code generation error.

238 illegal size of operand

Code generation error.

239 type should be LONGREAL

Value is too large to be expressed as a REAL.

240 parameter should be a dynamic array

Parameters for procedures such as:

```
PROCEDURE Test(x:ARRAY [1..10] OF CHAR);
```


are not allowed. the array should be declared as being open such as:

```
PROCEDURE Test (x:ARRAY OF CHAR);
```

241 illegal type for floating-point operation

REAL and LONGREAL values are not assignment compatible.

244 implementation restriction for floating-point comparisons

Code generation error.

250 invalid library specification for INLINE()

The library specification of an INLINE() procedure must be either a constant or a variable. It may not be an expression.

251 incorrect number of arguments for INLINE()

INLINE procedures require a minimum of 2 parameters: a library base specification, and a library base offset. As well, the number of arguments must be two larger than the number of parameters in the associated procedure.

252 invalid register number for INLINE()

This error occurs when an attempt is made to use a non-existent register number in an INLINE procedure definition. Valid register numbers range from 0 to 15 and are mapped to the D0-D7/A0-A7 68000 processor registers.

260 PROC, VAR or IMPLEMENTATION specified with \$I-

This error occurs when an attempt is made to use the \$I- compiler switch in a definition module which has some procedures or variables declared in it. This error will also be generated if an attempt is made to use the \$I- switch in an implementation module.

The \$I- switch is used to indicate that there is no implementation module associated with the current definition module. This can

only be true when the current definition module only has type and constants declarations, as well as `INLINE()` procedure definitions.

261 relocation buffer full

This error occurs when the compiler's relocation buffer overflows. The problem can usually be solved by increasing the size of the buffer. The relocation buffer is used to store information on every reference to an object which is external to the current compilation unit.

262 procedure result must be 1, 2, 4 or 8 bytes

This implementation restricts the type of values returned by a procedure call. No complex type such as arrays or records may be returned. To obtain such values from a procedure, either return a pointer to the object, or pass a VAR parameter of the required type to the procedure, and have the procedure manipulate the parameter.

269 body of FORWARD procedure never defined

This error occurs when a procedure which was previously declared as being FORWARD was never actually implemented in the current module.

270 opaque type must occupy 4 bytes

Opaque types always occupy four bytes of storage. This means that an opaque type cannot be implemented as a CARDINAL or an INTEGER, but can be implemented as LONGCARD, LONGINT, REAL or POINTER.

271 actual size not multiple of formal size

This error occurs when an attempt is made to pass an odd-length structure to an ARRAY OF WORD parameter of a procedure. In this case, make the parameter an ARRAY OF BYTE instead.

273 write to read-only \$\$- parameter

This error occurs when an attempt is made to assign a value to a

value parameter which was passed using the \$\$- compiler switch. three solutions are possible: Remove the compiler switch, make the offending parameter into a VAR parameter, or change the code so as not to modify the string parameter.

274 mismatch in module name specifications

This error occurs when the actual name of an imported module differs from the name specified in the import list of the current compilation unit. For example:

```
FROM inout IMPORT WriteString;  
(* Should be InOut *)
```

Return codes for support programs and utilities

The following list details the error codes each program will return in a given situation. These return codes can be tested for within batch files in order to determine whether or not the program executed successfully or not.

M2Errors

- 20 Bad arguments
- 10 Couldn't open window
- 5 No error log file

M2FastLoad

- 20 Bad arguments
- 10 Not enough memory
- 5 I/O error. Not able to write or read a file. Bad FAST file.

M2Settings

- 10 Not enough memory

M2Batch

- 20 Bad arguments
- 10 Not enough memory
- 5 I/O error. Not able to read or write a file

M2Prof

- 20 Bad arguments
- 10 Not enough memory, couldn't allocate TRAP
- 5 I/O error. Not able to read or write a file

NOTE: This program will not profile procedures which use the \$X- compilation switch. This includes 32-bit LONGCARD/LONGINT multiplication and division as well as REAL arithmetic.

B. Suggested reading

Since programming often comes down to a simple process of trial and error, having one or several of these excellent references on hand can be invaluable.

Modula-2 references

A guide to Modula-2

Kaare Christian

Springer-Verlag

An excellent book for people with some prior programming experience, interested in learning Modula-2.

Modula-2, A Seafarer's Guide and Shipyard Manual

Edward J. Joyse

Addison-Wesley

Highly recommended for novice Modula-2 programmers.

Modula-2, A Software Development Approach

Gary A. Ford & Richard S. Wiener

Wiley Press

A very well written highly technical book for advanced Modula-2 programmers.

Programming In Modula-2

Prof. N. Wirth

Springer-Verlag

This is a clear and concise reference to the language as written by the creator of Modula-2. It documents Modula-2's language specifications in detail, syntax and style. This book is of little or no value to beginners as it gives little or no tutorial information.

Modula-2 for Pascal Programmers

Cleaves, R.

Springer-Verlag.

Data Structures with Modula-2

Feldman, M.B.

Prentice Hall.

Invitation to Modula-2

Greenfield, S.B.

Petrocelli Books Inc.

Modula-2 Programming

Kaplan, I. & Miller, M.

Hayden Book Co.

Problem Solving and Structured Programming in Modula-2.

Koffman, E.B.

Addison-Wesley.

A Second Course in Computer Science with Modula-2.

McCracken, D.D. & Salmon.

J Wiley & Sons.

Modula-2: Constructive Program Development.

Messer, P. & Marshall, I.

Blackwell Scientific Publications.

Modula-2 Text and Reference

Moore, J.B. & McKay, K.N.

Prentice Hall.

A First Course In Computer Science with Modula-2.

Pinson, L. Sincovec, R. Wiener, R.

J Wiley & Sons.

Modula-2, Discipline and Design.

Sale, A.

Addison-Wesley.

Programming Expert Systems in Modula-2.

Sawyer, B. & Foster, D.

J Wiley & Sons.

Modula-2 Made Easy

Schildt, H.

McGraw-Hill.

Advanced Modula-2

Schildt, H.

McGraw-Hill.

Data Structures using Modula-2.

Sincovec, R.F. & Wiener, R.S.

J Wiley & Sons.

Data Structures and Abstract Data Types and Modula-2

Stubbs, D.F. & Webre, N.W.

Brooks/Cole.

A First Course In Programming with Modula-2.

Terry, P.D.

Addison-Wesley.

Amiga references

Amiga ROM Kernel Manual

Intuition: The Amiga User Interface

AmigaDOS Manual

Addison Wesley

These three books cover in-depth the Amiga's internal ROM, Intuition, and AmigaDOS functions.

Amiga Programmer's Handbook

Sybex Inc.

For the interested Amiga programmer. Contains detailed descriptions of the Amiga's operating system and the functions related to it.

C. Glossary

This appendix describes some of the terms used when discussing Modula-2 programming. We have not attempted to cover general computing terms or those specific to the Amiga. If you require further help in these areas, please consult a general purpose computing guide or the Amiga technical manuals.

ADDRESS

An address is the location of a piece of memory. In Modula-2 the type **ADDRESS** is used to hold values that can be the location of any part of the Amiga's memory.

ASSIGNMENT COMPATIBLE

To use the assignment operator `:=` the type of the source and the destination must be assignment compatible. This normally means that they are of the same type, or the size and the range of the two types are compatible.

BASE TYPE

Every subrange type has a **BASE TYPE** which is the type of its values. For example; in `ARRAY [0..10] OF CHAR` the subrange is `0..10` and the base type is `CHAR`.

BUFFERING

A technique in which input/output is not performed each time a read or write request is made, but rather is read or written to an intermediate buffer area.

COMPILATION UNIT

There are three type of compilation unit in Modula-2. Each contains all the information the compiler needs to produce its output file. The first is a **DEFINITION MODULE** which is translated into symbolic data. The second is an **IMPLEMENTATION MODULE** which must match a definition module's symbol file and is translated into machine code. The third is the **PROGRAM MODULE** which may not need a definition module and contains the statements first run when the program is executed.

DEFINITION MODULE

A compilation unit that contains the interface specifications of its associated implementation module.

DEREFERENCING

The operator '^' is called the dereferencing operator. Dereferencing is the act of accessing the item pointed to by a pointer type.

DYNAMIC ALLOCATION

Normal Modula-2 global variables are allocated by the compiler at compilation time. They are said to be static as they are fixed. Storage for variables that is performed at run time is said to be dynamic. It is most commonly used with pointer types.

EBNF

Stands for Extended Backus Naur-Formalism. This notation is used to describe the syntax of Modula-2 in a formal way.

FUNCTION PROCEDURE

A procedure that returns a value.

HIDDEN TYPE

See Opaque.

IMPLEMENTATION MODULE

A compilation unit that contains the implementation of the interface described by its definition module.

INDEX TYPE

The type of the subrange of an array structure. For example; in ARRAY [0..10] OF CHAR the index type is CARDINAL or INTEGER, you cannot use any other type to index the array.

LOCAL

Used when describing the scope of a variable. For example; a procedures variables are said to be local to the procedure. This term is also used to describe modules within an implementation module as they are local to the implementation module.

MODULE BODY

Used to describe the initialisation portion of an implementation or program module.

NIL

This name is given to a predefined item of type ADDRESS and the value 0. Note the value can be different on other implementations of Modula-2.

OPEN ARRAY

An array specification in which the index bounds of the array are not specified. For example; PROCEDURE x (VAR a : ARRAY OF CHAR). The array 'a' is said to be an open array.

OPAQUE

Used when describing the export of a type. If the type is not fully specified in the definition module (eg. TYPE File ;) it is said to be opaque as any modules importing the type cannot 'see' what type it is. This is also sometimes referred to as a 'hidden' type.

PROCEDURE BODY

The executable portion of a procedure.

QUALIFIED

An identifier is termed qualified when the module name is used as a prefix. For example; InOut.WriteString.

TYPE COERSION

The compiler can be instructed to treat a variable as a type different from the type it was declared with. This is termed type coercion. It should not be confused with type conversion, as the data is not changed, just treated differently.

TYPE CONVERSION

The act of converting data from one type to another.

D. Technical support

By returning your registration card, you can obtain technical support from M2S.

In addition, users with modems can contact members of the technical support team in the M2S conference on BIX, the Byte Information Exchange, or the AmigaVendor forum on Compuserve.

Checklist

Before requesting Technical Support, take a minute to read over the points listed below. Doing so may solve your problem or answer your question. Also, the Support Specialist will be able to help you more quickly if you have this information handy.

- Computer Type/Model
 - Amount of memory
 - Hard Disk
 - Operating system version
 - Printer
 - List of expansion board products in use
1. **Check to see if your hardware meets the minimum system requirements for the software you are using. These requirements are listed in the documentation (memory, disk storage, operating system and version).**
 2. **Check to make sure that your hardware and peripherals are set-up according to the documentation and that all cable connections are secure.**

4. If your question is not answered by the documentation, contact M2S. A Specialist will assist you in solving the problem.

Joining BIX for technical support

As an M2S customer, you can use the special "keyword" we've included with these instructions and pay a one-time BIX registration fee of just \$25. That's \$14 off the regular fee of \$39. Once you join, you'll also be automatically joined to a free learn conference which shows you how to use BIX.

There are a number of advantages for you and for us in online product technical support:

First, we can post answers to many of the most commonly asked questions about our products online. Easy-to-use BIX search commands let you quickly check to see whether an answer to your particular question is already available.

Secondly, you can make efficient use of your time...and ours and get the information you need to get the most out of our products and your Amiga system. Log on to BIX, leave your questions in our conference and log off. No busy signals or waiting on hold. BIX can handle hundreds of simultaneous callers. Next time you log on, if responses have been posted to your questions, BIX will notify you and show you those responses automatically.

Finally, by joining our conference, you are brought together with other users of M2S products from around the world. Users often help each other, and share information on their use of our products which can help you get more out of them.

Scores of hardware and software companies, including Commodore and many other Amiga vendors, are now using BIX for online technical support. In addition, you can select from more than 160 conferences where users with similar interests share information, opinions and ideas on everything from computers, operating systems and programming languages to artificial intelligence, chips and graphics. You also get access to a daily newswire of industry developments and information, timely general interest business news, extensive new product listings of hardware and software, a large library of shareware software you can download, a real time chat facility and electronic mail as part of your membership. You pay only connect charges.

You can join BIX right now with American Express, VISA or MasterCard payment. Other payment options are available. Call the BIX Helpline for more information.

BIX online signup procedures:

1. **Set your telecommunications program for either full duplex, 8-bit characters, no parity, 1 stop bit; or 7-bit, even parity, 1 stop. Call at either 300 or 1200 baud.**
2. **Dial the local Tymnet number.**

Most callers can reach BIX via Tymnet, a low-cost network with local phone numbers throughout the U.S. and major Canadian cities. Call 1-800-336-0149 to get the local Tymnet number for your area. For information on accessing BIX from countries outside the USA or Canada, call or write BIX.

3. **Log in as follows:**

Tymnet Prompt

You Enter

Garbled characters or request to
enter terminal identifier

a

log in

BIX
press RETURN

Name:

vs.M2S
press RETURN

4. **Complete the BIX registration.**

When logging in be sure to use the above keyword "vs.M2S" so you'll get the special low registration fee. Once you've registered join the free learn conference and then join us with the command:

`join M2S`

BIX will send you a user manual and subscriber agreement. Sign and return one copy of the agreement to BIX.

Need more help?

If you would like more information about BIX before you decide, contact M2S Technical Support or contact BIX directly at:

One Phoenix Mill Lane
Peterborough, NH
03458

Tel: (800) 227-2983 (free inside Canada and the US)
(603) 924-7681 (for New Hampshire residents and elsewhere)

BIX lines are open from 8:30 AM to 11 PM Eastern time weekdays.

Compuserve

M2S also has a section in the AmigaVendor forum of the Compuserve information service. Once logged onto Compuserve, type 'GO AMI-GAVENDOR', then select message section 12. Our Compuserve ID is 76004,2054.

For details of how to join contact Compuserve direct on (800) 848-8990.

E. Customizing M2Data files

The M2Data: drawer contains several files used by the M2Sprint compiler, linker and editor to control various features. This appendix describes every one of those files and explains what changes you can make to them in order to tailor M2Sprint's environment to your particular needs.

M2Data contents

The M2Data drawer contains the following files:

M2.caserecorrect	Used by the editor for the Correct Case feature
M2.compconfig	Compiler configuration
M2.editconfig	Editor configuration
M2.errorlist	Descriptions of every Modula-2 error, used by the editor
M2.link	Icon image used by the compiler
M2.linkconfig	Linker configuration
M2.prog	Icon image used by the linker
M2.searchpath	Search path used by the compiler and linker
M2.wordcomplete	Used by the compiler for the Complete Word feature

M2.caserecorrect

This file is necessary to support two of the special features of the M2Sprint editor: Correct Case and Correct Word.

M2.caserecorrect contains a list of words. These words are used by the editor when performing case correction or when correcting the spelling of words. The file is straight ASCII and may be edited using the M2Sprint editor.

As shipped, M2.caserecorrect contains all of the standard Modula-2 keywords. This allows M2E to convert all of the M2 keywords (such as PROCEDURE) to their correct case as you type them in, or to correct typos in them when you select Correct Word. The editor only knows the words in this file.

You can easily modify M2.caserecorrect to contain any words you want. You could enter Amiga-specific terms such as "Intuition" or "BltMaskBitMapRastPort", so that you can type these words in the editor, and have M2E transform them to the correct case, or correct typos.

The case of the words you enter in the file is important as they will be used verbatim for Case Correction. By default, all words in the file are in CAPS, since all Modula-2 keywords are in caps. However, you can enter words in either upper or lower case, and Correct Case will correctly transform the words you type in the editor to match what is in M2.caserecorrect. As well, all words you enter in the file can be used with Correct Word.

To activate the changes you make to the M2.casecorrect file, you must first save the new file in M2Data:, then quit and reload the editor. This forces M2E to reload it's data files.

NOTE: If the editor is loaded using the HOT option, it needs to be completely removed from memory using the CTRL-ALT-Q hot key combination. See Chapter 11 - "M2E - The editor" for more information.

M2.compconfig

The file is used by the compiler to determine its default buffer sizes, and switch settings. You can alter these values by using the "Compiler" section of the M2Settings requester, or by using the stand-alone M2Settings program.

M2.editconfig

This file is managed by the M2Sprint editor. It contains configuration information that the editor uses when first started. The file contains the following:

- default settings for the menu items in M2Config menu
- default setting for the auto save delay
- default setting for the TAB width
- default setting for the right margin
- default definitions for the 10 string macros
- default definitions for the 10 ARexx macros
- default screen coordinates of the main M2E editing window
- default screen coordinates of all the M2E requesters
- default settings for the toggle gadgets in the Find and Find/Change requesters

- values for the "Program" section of the M2Settings requester.

To modify any of these defaults, simply start the editor and adjust all the values as you like them and choose "Config/Save Config".

The config file will retain the current editor window coordinates and size as the default. This means that after you save the configuration, the next time you start the editor, the initial size of the main editing window will be the same as when you saved the configuration settings. This feature is quite valuable for European users, or users wishing to use the editor in Interlace mode. To change the default window size and position, simply start the editor and resize the window to the appropriate dimension, and save the configuration. The next time you start the program, the initial window will have the correct size.

M2.errorlist

This file contains a one line description of every error that the M2Sprint compiler can generate. Both the M2Sprint editor and the M2Errors utility use this file to present English descriptions of compilation errors. By using the MakeErrorList utility you can alter the contents of this file. This may be useful when trying to customize the environment for various languages such as French, German, and others.

M2.link

This file is used as template when the compiler generates an icon for an object file.

Depending on the setting of the "Config/Save Icons" menu item, the compiler may generate icons to go with every .lnk file it creates. When it comes time to make an icon, the compiler will look for the file called "M2.link", and will output a copy of this file with the name of the .lnk file.

By replacing M2.link by one of your own icons, you will cause the compiler to generate icons with the same appearance as yours.

M2.linkconfig

The file is used by the linker to determine its default switch settings, RunTime module name, and IMG file names. You can alter these values by

using the "Linker" section of the M2Settings requester, or by using the stand-alone M2Settings program.

M2.prog

This file is used as template when the linker generates an icon for an executable

Depending on the setting of the "Config/Save Icons" menu item, the linker may generate icons to go with every executable file it creates. When it comes time to make an icon, the linker will look for the file called "M2.prog", and will output a copy of this file with the name of the executable file.

By replacing M2.prog by one of your own icons, you will cause the linker to generate icons with the same appearance as yours.

M2.searchpath

This file is used by both the compiler and linker. It contains a series of lines describing where the two programs are to look when searching for Modula-2 .lnk or .sym files.

When compiling, the compiler will often need to access special .sym files. Typically, the compiler will look for the files in the current drawer, if not found, it will then look in the M2: drawer. If still not found, it will present a file requester prompting the user to select the file. By using M2.searchpath, you can specify other drawers to be searched before presenting the file requester.

As an example, if you have most of your .sym files in a RAM disk to speed up compiling, but have not enough room to store all of them there. You could copy only the more used files to the ram disk and leave the less commonly referenced files on disk. You would then use M2.searchpath to tell the compiler that certain files are on disk. M2.searchpath might look like `RAM:MainM2 DH0:ExtraM2.`

With a M2.searchpath file defined as above, the compiler would first try to find .sym files by looking in the current drawer. If it's not there, then it would look in the M2: drawer. If still not found, then it would look in RAD:MainM2, and finally in DH0:ExtraM2.

Every line of the file indicates a specific search path. The path specification may be relative to the current drawer.

M2.wordcomplete

This file is necessary to support the special Complete Word feature of the M2Sprint editor.

M2.wordcomplete contains a series of lines describing all of the expressions that M2E can accept in its Complete Word function. Every line contains a single expression which the editor uses when trying to complete a word. You can add or delete entries in this file to make M2E understand your own set of expressions. The file is straight ASCII and may be edited using the M2Sprint editor.

To explain the format of the M2.wordcomplete file, it is best to use an example. Using the default M2.wordcomplete file, if you type "PROCE" and hit the F7 key from within M2E, you will obtain the following:

```
PROCEDURE (); VAR  
  
BEGIN  
  
END ;
```

The cursor will be positioned on the open parenthesis after the word PROCEDURE . Looking at the M2.wordcomplete file, you can see an entry for the word PROCEDURE which looks like:

```
PROCEDURE @();\nVAR\n\nBEGIN\n\nEND ;\n
```

The relation between the line in M2.wordcomplete and the result on screen is simple.

First, there are the "\n" combinations. These indicate that there is a line break at that position in the template. So basically, when inserting the text in the editor, a "\n" combination will yield the same result as hitting the RETURN key. This is why the words VAR and BEGIN, for instance, end up on two separate lines in our example.

Second, the "@" symbol in the M2.wordcomplete line, indicates where the cursor should be positioned after the word has been completed. The cursor is positioned on the character following the "@" sign. In our example, the cursor get positioned on the open parenthesis, ready to let the user enter the procedure name.

To further explain this format, let us take a template and turn it into the necessary sequence of characters to include in the word complete file.

If we have a template that looks like:

```
WHILE ( ) DO  
  
END ;
```

We need to specify every line of the template in the M2.wordcomplete file. Instead of hitting RETURN after every line though, we use the "\n" combination:

```
WHILE ( ) DO\n\nEND;\n
```

Note that to specify the empty line, an extra "\n" was added after the word DO.

Now given our template, where would it be more efficient to position the cursor after the template has been expanded in the user's text file? The logical choice is to position the cursor to let the user enter the boolean condition in between the parenthesis. This means that the cursor must be positioned on the closing parenthesis. To specify this, we include the "@" sign just before the parenthesis. This would turn our template into:

```
WHILE (@) DO\n\nEND;\n
```

You can customize M2.wordcomplete to allow you to use the special Complete Word feature to edit ARexx, "C" or Assembler files for example.

F. Run-time license

Programs developed using M2Sprint are stand-alone native code programs that do not require the M2Sprint system to run. Original purchasers who have agreed to the terms outlined in the "Program License Agreement" are free to develop and market products written in M2Sprint without any run-time license charge. We do require that you include the following reference in the manual or "About..." requester of the product.

**Developed using M2Sprint for the Amiga
M2S Inc., Dallas, Texas**

G. M2E ARexx

command summary

COMMAND

Center - Center the current line or block

RESULT

Always returns 0

PURPOSE

This performs the same action as the Extras/Center Line/Block menu item. The centering is performed according to the current right margin and with the left edge of the window.

COMMAND

Clear - Erase the current file from memory

RESULT

0 if the file was cleared

1 if the file was not cleared

PURPOSE

This performs the same action as the Project/Clear menu item. If the current document has changed, a requester will appear asking for confirmation. If the user chooses not to clear the document, then this will return 1.

COMMAND

Compile - Compile the currently loaded Modula-2 program.

RESULT

0 if compilation successful

1 if there were some compilation errors

5 if the user aborted the compilation with CTRL-C

10 if there was not enough memory

PURPOSE

This performs the same action as the Modula-2/Compile menu item.

COMMAND

Complete_Word - Complete the current word.

RESULT

Always returns 0

PURPOSE

This performs the same action as the Line/Word/Complete Word menu item

COMMAND

Copy - Copy the currently selected block to the clipboard

RESULT

- 0 if successful
- 1 if the clipboard returned an error
- 10 if there was not enough memory

PURPOSE

This performs the same action as the Edit/Copy menu item.

COMMAND

Correct_Word - Correct the current word

RESULT

Always returns 0

PURPOSE

This performs the same action as the Line/Word/Correct Word menu item.

COMMAND

Count_Words_To_EOF - Count the number of words from the current location to the end of file

RESULT

Returns the number of words

PURPOSE

This performs the same action as the Extras/Count Words To EOF menu item.

COMMAND

Cut - Cut the currently selected block to the clipboard

RESULT

- 0 if successful
- 1 if the clipboard returned an error
- 10 if there was not enough memory

PURPOSE

This performs the same action as the Edit/Cut menu item.

COMMAND

Delete - Delete a character from the current file

RESULT

- 0 if successful
- 1 if the cursor was at EOF and the operation could not be performed
- 10 if there was not enough memory

PURPOSE

This performs the same action as hitting the DEL key.

COMMAND

Down - Move the cursor down one line.

RESULT

- 0 if successful
- 1 if the cursor was at EOF and the operation could not be performed

PURPOSE

This performs the same action as hitting the down arrow cursor key.

COMMAND

DownCase_Block - Convert the characters in the selected block to lower case

RESULT

Always returns 0

PURPOSE

This performs the same action as the Edit/DownCase menu item.

COMMAND

DownCase_Word - Convert the current word to lower case

RESULT

Always returns 0

PURPOSE

This performs the same action as the "Line/Word/DownCase Word" menu item.

COMMAND

EOF - Move the cursor to the end of the file

RESULT

Always returns 0

PURPOSE

This performs the same action as hitting ALT + Down arrow.

COMMAND

EOL - Move the cursor to the end of the current line

RESULT

Always returns 0

PURPOSE

This performs the same action as hitting ALT + Left arrow.

COMMAND

Erase - Delete the currently selected block

RESULT

Always returns 0

PURPOSE

This performs the same action as the Edit/Erase menu item.

COMMAND

Find [string] - Search through the current file for a string.

RESULT

0 if string found
1 if string not found

PURPOSE

You supply a search string, and this command will search the current file for you. If the string is found, the cursor will be positioned on the starting character of the found text. If the string is not found, then the cursor position remains unchanged.

COMMAND

Format_Paragraph - Reformat a paragraph

RESULT

Always returns 0

PURPOSE

This performs the same action as the "Extras/Format Paragraph" menu item.

COMMAND

Get_Config [variable] - Read the current config settings

RESULT

See below

PURPOSE

This command serves to read most of the configuration parameters of the editor. [variable] defines which setting to read, For toggle settings, the return value will either be 0 (OFF) or 1 (ON). All other return values are numeric. The supported settings are:

ARexx_Console	returns 0 or 1
Auto_Indent	returns 0 or 1
Auto_Save	returns 0 or 1
Correct_Case	returns 0 or 1
Give_Warnings	returns 0 or 1
Make_Backups	returns 0 or 1
Overstrike	returns 0 or 1
Save_Icons	returns 0 or 1
Status_Bar	returns 0 or 1
TABs_Give_Spaces	returns 0 or 1
Word_Wrap	returns 0 or 1
Auto_Save_Delay	returns the current auto save delay
Right_Margin	returns the current right margin
TAB_Width	returns the current TAB width
Backward	returns 0 or 1
UPPER_lower	returns 0 or 1
Whole_Word	returns 0 or 1

COMMAND

Goto_Last_Modified - Move the cursor to the position where modifications were last made

RESULT

Always returns 0

PURPOSE

This performs the same action as the "Line/Word/Goto Last Modif." menu item.

COMMAND

Hex_Display - Open the Hex Display window

RESULT

Always returns 0

item.

COMMAND

New_And_Open - Open a new editor window and pop up the file requester

RESULT

Always returns 0

PURPOSE

This performs the same action as the "Project/New & Open" menu item.

COMMAND

Next_Word - Move the cursor to the start of the next word

RESULT

- 0 if successful
- 1 if the cursor was at EOF and the operation could not be performed

PURPOSE

This performs the same action as hitting the right arrow cursor key while holding down the SHIFT key.

COMMAND

Open [name] - Load a new file in the editor

RESULT

- 0 if successful
- 1 if the file could not be found
- 10 if there was not enough memory to load the file

PURPOSE

Tries to load [name] in the editor. If the current document has changed, a requester will appear asking for confirmation. If the user chooses not to open the document, then this will return 1.

COMMAND

Open_Clip [name] - Load a text file directly into the clipboard

RESULT

- 0 if successful
- 1 if the file could not be found
- 10 if there was not enough memory

PURPOSE

Loads [name] directly into the clipboard.

COMMAND

Paste - Insert the clipboard contents at the current cursor position.

RESULT

- 0 if successful
- 10 if there was not enough memory

PURPOSE

This performs the same action as the "Edit/Paste" menu item

COMMAND

Prev_Word - Move the cursor to the start of the previous word

RESULT

- 0 if successful
- 1 if the cursor was at the start of the file and the operation could not be performed

PURPOSE

This performs the same action as hitting the left arrow cursor key while holding down the SHIFT key.

COMMAND

Print - Print the current file

RESULT

- 0 if successful
- 1 if failure

PURPOSE

This performs the same action as the "Project/Print" menu item.

PURPOSE

This performs the same action as the "Extras/Show Hex Display..." menu item.

COMMAND

Jump_To_Line [line #] - Move the cursor to a specific line number.

RESULT

- 0 if the cursor was correctly positioned
- 1 if the line number requested is not part of the current file

PURPOSE

This performs the same action as the "Line/Word/Jump To Line..." menu item.

COMMAND

Justify - Justify the current line or block

RESULT

Always returns 0

PURPOSE

This performs the same action as the Extras/Justify Line/Block menu item. The justifying is performed according to the current right margin and with the left edge of the window.

COMMAND

Left - Move the cursor left one character.

RESULT

- 0 if successful
- 1 if the cursor was at the start of the file and the operation could not be performed

PURPOSE

This performs the same action as hitting the left arrow cursor key.

COMMAND

Link - Link the file currently specified in M2Settings

RESULT

- 0 if linking successful
- 1 if there were some linking errors
- 10 if there was not enough memory

PURPOSE

This performs the same action as the Modula-2/Link menu item.

COMMAND

Mark Block - Turn on selection mode

RESULT

Always returns 0

PURPOSE

This performs the same action as the "Edit/Mark Block" menu item.

COMMAND

Match Bracket - Find a matching bracket

RESULT

- 0 if a match was found
- 1 if there were no matching bracket

PURPOSE

This performs the same action as the "Extras/Match Bracket" menu item.

COMMAND

New Window - Open a new editing window

RESULT

Always returns 0

PURPOSE

This performs the same action as the "Project/New Window" menu

COMMAND

Print_Clip - Print the current contents of the clipboard

RESULT

- 0 if successful
- 1 if failure

PURPOSE

This performs the same action as the "Edit/Print Clip" menu item.

COMMAND

Recenter_Display - Puts the cursor in the middle of the display

RESULT

Always returns 0

PURPOSE

This performs the same action as the "Line/Word/Recenter Display" menu item.

COMMAND

Refresh_Display - Redraws the display window

RESULT

Always returns 0

PURPOSE

While a macro is executing, the display is not updated to help execution speed. When the macro terminates, the display is updated automatically. If you need to update the display while a macro is running, you can use this command.

COMMAND

Right - Move the cursor right one character.

RESULT

- 0 if successful
- 1 if the cursor was at EOF and the operation could not be performed

PURPOSE

This performs the same action as hitting the right arrow cursor key.

COMMAND

Save - Save the current file

RESULT

0 if successful
1 if failure

PURPOSE

This performs the same action as the "Project/Save" menu item.

COMMAND

Save_As [name] - Save the current file under a different name

RESULT

0 if successful
1 if failure

PURPOSE

This performs the same action as the "Project/Save As..." menu item.

COMMAND

Save_Clip [name] - Save the current contents of the clipboard to a file

RESULT

0 if successful
1 if failure

PURPOSE

This performs the same action as the "Edit/Save Clip" menu item.

COMMAND

Save_Config - Save the current configuration settings

RESULT

Always returns 0

PURPOSE

This performs the same action as the "Config/Save Config" menu item.

COMMAND

Screen_Down - Move the cursor down one window.

RESULT

- 0 if successful
- 1 if the cursor was at EOF and the operation could not be performed

PURPOSE

This performs the same action as hitting the down arrow cursor key while holding down the SHIFT key.

COMMAND

Screen_Up - Move the cursor up one window.

RESULT

- 0 if successful
- 1 if the cursor was at the start of the file and the operation could not be performed

PURPOSE

This performs the same action as hitting the up arrow cursor key while holding down the SHIFT key.

COMMAND

Set_Config [variable] [value] - Adjust the current configuration settings

RESULT

Always returns 0

PURPOSE

This command serves to adjust most of the configuration parameters of the editor. [variable] defines which setting to modify, and [value] is the new value for this setting. For toggle settings, [value] corresponds to either ON or OFF. Otherwise, you supply a number. The supported settings are:

ARexx_Console	[ON OFF]
Auto_Indent	[ON OFF]
Auto_Save	[ON OFF]
Correct_Case	[ON OFF]
Give_Warnings	[ON OFF]
Make_Backups	[ON OFF]
Overstrike	[ON OFF]
Save_Icons	[ON OFF]
Status_Bar	[ON OFF]
TABs_Give_Spaces	[ON OFF]
Word_Wrap	[ON OFF]
Auto_Save_Delay	[minutes]
Right_Margin	[character count]
TAB_Width	[character width]
Backward	[ON OFF]
UPPER_lower	[ON OFF]
Whole_Word	[ON OFF]

COMMAND

SOF - Move the cursor to the start of the current file.

RESULT

Always returns 0

PURPOSE

This performs the same action as hitting ALT-Up Arrow.

COMMAND

SOL - Move the cursor to the start of the current line.

RESULT

Always returns 0

PURPOSE

This performs the same action as hitting ALT-Left arrow.

COMMAND

Spaces_To_TABs - Convert spaces to TAB characters.

RESULT

Always returns 0

PURPOSE

This performs the same action as the Extras/Spaces to TABs menu item.

COMMAND

Swap_Chars - Exchange the character under the cursor with the one to the left.

RESULT

Always returns 0

PURPOSE

This performs the same action as the Line/Word/Swap Chars menu item.

COMMAND

TABs_To_Spaces - Convert all TAB characters to spaces.

RESULT

Always returns 0

PURPOSE

This performs the same action as the Extras/TABs to Space menu item.

COMMAND

Text [string] - Insert a string in the current file

RESULT

0 if successful

10 if there was not enough memory

PURPOSE

This is the same as typing the given string on the keyboard

COMMAND

Toggle_Char_Case - Toggle the case of the character under the cursor

RESULT

Always returns 0

PURPOSE

This performs the same action as the "Line/Word/Toggle Char Case" menu item.

COMMAND

Unload_Comp_Link - Free the memory used by the compiler and linker.

RESULT

Always returns 0

PURPOSE

This performs the same action as the Modula-2/Unload Compile-Link menu item.

COMMAND

Up - Move the cursor up one line.

RESULT

- 0 if successful
- 1 if the cursor was at the start of the file and the operation could not be performed

PURPOSE

This performs the same action as hitting the up arrow cursor key.

COMMAND

UpCase_Block - Convert the characters in the selected block to upper case

RESULT

Always returns 0

PURPOSE

This performs the same action as the Edit/UpCase menu item.

COMMAND

UpCase Word - Convert the current word to upper case

RESULT

Always returns 0

PURPOSE

This performs the same action as the "Line/Word/UpCase Word" menu item.

.DEF

11-6, 11-40

.MOD

11-6, 11-40

AREXX

9-10, H-1

ARP

4-1, 11-6, 11-28, 9-1, 9-9

ARexx

1-1, 11-1, 11-17, 11-18, 11-37, 11-42

ASCII

11-22, 11-37

Activation Flags

17-10, 17-11, 17-12, 17-13

Amiga 2000

5-3

AmigaDOS

11-6, 11-28

Animation

16-1, 16-2, 19-1

Audio

15-1

Auto-Indent

11-16, 11-25, 11-42

Auto-Save

11-8

BIX

D-3

BOB

19-6, 19-7, 19-8

BOBS

19-1, 19-2

Backup

5-1, 5-3, 11-9, 11-42

Batch Compilation

8-5, 11-40

Bit-Planes

16-1, 16-9, 19-13, 26-8

Bitmap

16-2, 19-1, 24-2, 24-5, 25-1, 25-4

Blitter

19-1

Bookmarks

11-21, 11-24

CHIP

19-7, 19-11

CLI

1-1, 8-2, 8-6, 8-7, 8-10, 8-11, 5-3, 5-5, 11-2, 11-3, 11-6,
11-39, 11-47

CLI/Workbench

11-46

Case-Correct

11-42

Change

11-35, 11-36

Clear

11-29

Clipboard

11-13, 11-14, 11-15, 11-31, 11-32

Close

11-29, 11-30

Close Gadget

11-4

Collision

19-1

Color

16-12, 17-18, 17-19, 17-23, 17-25, 19-11, 21-25, 21-26,
22-3, 24-4, 24-6, 28-3, 28-4

Color Register

16-1

Colors

25-4, 26-9

Commands

2-4

Compiler Switch

11-44

Complete Word

11-1, 11-19, 11-20, 11-26, 11-33

Configuration

11-17

Copy

11-12, 11-13, 11-15, 11-24, 11-31

Correct Case

11-1, 11-19, 11-25

Correct Word

11-19, 11-20, 11-21, 11-26, 11-33

Cursor

8-3, 11-4, 11-10, 11-23, 11-31, 11-33, 11-34

Cut

11-12, 11-13, 11-15, 11-25, 11-31

Debug

11-44, 11-45

Debugger
11-44

Debugging
8-7, 11-1

Definition Modules
8-5

Drawer
11-27, 11-28

Erase
11-25, 11-31

Error
11-41

Errors
8-7

File Requester
11-7, 11-27

Find
11-25, 11-34, 11-36

Font
16-8, 17-17, 21-24, 26-7

Gadget
9-4, 17-5, 17-6, 17-26, 17-27, 17-28, 17-29, 17-30, 17-31,
17-32, 17-33, 17-34, 23-3, 23-4

Gadgets

11-5, 17-1, 20-1, 23-1

Gels

19-1, 19-9

HAM

22-4

HOT

11-24

HOT Switch

11-3

Heap

11-45

Hex

11-22, 11-25, 11-38

Highlighting

11-12

Hot Key

11-3

IDCMP

20-1, 9-5

IFF

24-1, 24-3, 24-4, 24-5, 25-1, 25-3, 25-4, 9-1, 9-8

IFF2Obj
11-46, 17-29

ILBM
24-1, 24-4, 24-5, 25-1, 25-4

IMG
11-46

Icon
8-4, 8-12, 5-3, 11-2, 11-7

Icons
8-6, 11-43

Image
17-29, 19-3, 19-7

Implementation Modules
8-5

Insert
11-15

Intuition
16-1, 17-1, 17-10, 17-11, 17-12, 17-13, 17-31, 20-1, 21-1,
21-21, 21-22, 21-23, 22-1, 23-1, 26-1, 26-8, 28-1, 28-5

Joystick
18-1, 18-4, 18-5

Keyboard
2-4, 11-10

Kickstart

4-1

Library

1-2

Library Module Reference

2-1

Linking

8-9

M2A

11-3, 11-7

M2C

8-5

M2Data

11-17

M2E

8-2, 11-1

M2S

3-1, 5-2, 5-7

Macro

11-25

Macros

11-24, 11-37

Mark

11-12, 11-26, 11-31

Master Diskettes

5-1, 5-3

Menu

2-4, 11-24, 21-1, 21-2, 21-5, 28-3, 28-4

Menus

20-1

Mouse

8-3, 11-10, 11-11, 11-26, 11-31, 18-1, 20-1

Mouth

27-11, 27-12

Moving Text

11-13

Multitasking

11-2, 20-1

NTSC

16-1, 16-9, 26-8

Narrator

27-1, 27-3, 27-4, 27-9, 27-10, 27-11

New

11-29

New Window

11-2, 11-25

Open

11-6, 11-7, 11-25, 11-28, 11-29, 11-32, 26-8

Overstrike

11-15, 11-25

PAL

11-17, 16-1, 16-9, 26-8

Paste

11-12, 11-14, 11-15, 11-25, 11-31, 11-32

Pen

16-4, 16-5, 16-12, 17-18, 17-19, 17-23, 17-25, 21-25,
21-26, 26-3, 26-4, 26-9, 28-3, 28-4

Phoneme

27-1, 27-4

Pitch

27-8

Port

18-4

Preferences

11-9, 22-3

Print

11-29, 11-32

Printer

22-1

Printing
11-9, 11-14

Priority
15-1, 15-7

Programmer's Guide
2-1

RAM
4-1

READ_ME
5-1

ROM Kernel Manual
15-1, 19-2, 19-5, 19-6

Reduce Gadget
11-4

Requesters
11-5

Run
11-41

RunTime
11-46

Running
8-11

Save

11-15, 11-17, 11-24, 11-25, 11-28, 11-29, 11-32

Saving

11-8

Saving Files

8-3

Screen

9-5, 16-1, 16-2, 16-3, 16-4, 16-5, 16-7, 16-8, 16-9, 16-10,
16-11, 16-12, 19-3, 22-2, 22-3, 24-1, 24-4, 24-6, 26-1,
26-3, 26-4, 26-5, 26-8, 26-9

Scroll Bar

8-3, 11-5, 11-7, 11-10, 11-11, 11-27, 23-1

Scroll Bars

17-31

Scrolling

11-11

Selecting Text

11-12

Shift Text

11-13

Shift-Clicking

11-7

Sizing Gadget

11-5

Sound

15-1

Speech

27-1, 27-14, 27-15, 27-16

Sprites

19-1

Status Bar

8-4, 8-7, 11-5, 11-20, 11-21, 11-34, 11-43

TAB

11-9, 11-39, 11-43

TYMNET

D-4

Talk

27-13, 27-14

Title Bar

11-4

Translator

27-1, 27-3

Undo

11-25, 11-31

VSprite

19-1, 19-5, 19-11, 19-12

Volume
11-27, 11-28

Warranty
3-1, 5-7

Window
11-4, 11-29, 17-2, 20-1, 20-3, 21-2, 21-5, 22-2, 23-3, 28-1

Window Depth
11-4

Word Wrap
11-16, 11-25, 11-43

Workbench
1-1, 4-1, 8-2, 8-4, 8-6, 8-8, 8-10, 8-12, 5-3, 11-3, 11-6,
11-7, 22-1

Write Protect
5-3